前　言

早期的J2EE平台推出的EJB规范是为了简化分布式应用的开发，并在J2EE平台上为各种企业级服务提供最佳实践。相对于更早以前的诸如CORBA等分布式架构来说，EJB确实使分布式架构得以简化。在当时特定的历史条件下，基于EJB架构的企业级应用的确有其先进性。但是，随着时间的推移，基于EJB架构的企业级应用逐渐暴露出各种形式的不足。于是开发者们开始探索一种新的技术，希望这种技术能弥补EJB的缺陷，一场技术革命就此拉开了序幕。

实际上，我们可以从两个方面来看待EJB被“革命”的原因。首先，分布式架构更适用于那些大型的企业级应用，这些应用往往对安全性、可扩展性的要求更高。对于这类大型企业级应用来说，为了获得更重要的功能特性，分布式架构在性能等方面的缺陷是可以容忍的。然而，对于大多数基于J2EE平台的中小型企业级应用来说，分布式架构在性能方面的损失则是完全不可接受的，分布式架构的引入反而会增加开发和维护的难度，甚至导致整个项目失败。颇有讽刺意味的是，恰恰是开发者们早期对EJB架构的过度推崇造成了如今这种难以收拾的局面。EJB本身笨重的编程模型被应用到不合适的场景中，给积重难返的EJB压上了“最后一根稻草”。

其次，EJB规范本身可看作是为J2EE平台上的各种企业级服务提供的一种最佳实践，但是这一最佳实践实际上并未达到预期效果。实体Bean（Entity Bean）被证明是一种失败的实践，其他种种现象也从各个侧面反映了EJB的缺陷，如整个应用需要绑定到应用服务器、编程模型复杂而笨重、应用本身不易测试等。

随着EJB自身所暴露的缺陷越来越多，开发者们不再盲目崇拜EJB，而开始对它感到失望。在这种情况下，一场革命在所难免，而燃起这场革命的星星之火的就是Spring。Spring框架提倡一切从实际出发，使用基于POJO的轻量级编程模型推进整个应用的快速开发。通过使用IoC、AOP等先进的技术理念，结合对原有J2EE平台各种企业级服务的抽象和适度集成，Spring框架为Java平台上的企业级开发注入了新鲜的血液。

从Spring框架身上，纵观，你几乎可以看到整个基于Java平台的软件开发的演变历史；横看，你可以跟踪和捕捉当前业界最先进的理念和软件开发模式。如果是初次踏上Java平台这片领地，沿着Spring框架所展示的路线，你将迅速地领略到整个Java平台的强大魅力；如果你已经在Java平台“征战”多年，Spring框架一定能使你回忆起昔日所经历的“酸甜苦辣”，并更加深切地感受到Spring框架给我们带来的价值。

本书的主要目的就是和读者朋友一起以轻松愉快的心情饱览整个Spring框架的“无限风光”，让你能够在愉快的旅途之后以全新的状态投入到现实的Java企业级应用开发工作中去。

关于本书

促使我下决心写下这本书的原因很多，但主要可以归结为两点。一个是因为自身在经年的软件开

VISIT…

LANZAROTE Caliente.COM

发过程中的所见所闻，当历尽千辛走到Spring阳光下的时候，再回头看看，会发现自己和他人走过的路是多么地泥泞和曲折，而这些现在是完全可以避免的。可是，就算是在2007的项目中，我却依然发现大部分人还是在过去的沼泽中挣扎，所以我想通过这本书传达一种信息，希望人们能够尽快的跳出那片早就应该远离的沼泽。另一个原因则是希望通过对Spring框架的介绍，为大家传达一种业界内先进的理念和开发模式，为国内软件开发过程的改进和开发质量的提高，尽自己的一份力。

帮助你了解如何使用Spring框架所提供的各种方案来解决开发过程中所遇到的实际问题固然重要，但从Spring框架中汲取更多的"营养"才是我们应该进一步关注的，知其然，知其所以然，然后我们才可以在Java开发领域内游刃有余，Spring框架中所展现的开发理念以及方法学等，都应该成为我们日常开发过程中不可或缺的伙伴。所以，本书除了会像其他Spring书籍一样，对Spring框架提供的各种功能和特性进行详尽的讲解，中间还将穿插Spring实现中牵扯的设计模式和最佳实践的分析，以期最大限度地挖掘Spring这座宝藏。

技术是发展的，但思想却是延续的，当我们借助Spring之船飞渡波澜不惊的水面亦或闯过惊涛骇浪之后，不要忘记，或许某天我们会踏上另一条更好的船舶，但Spring之船所带给我们的那些理念和思想，却应该始终陪伴着我们，激励我们也好，鞭策我们亦善……

作者在线

如果在阅读本书的过程中遇到任何疑问或发现任何问题，可通过我的个人网站http://www.spring21.cn/与我取得联系。我会悉心为你解答有关本书的任何问题，虽然这是无偿的，但是我很乐意这样做，因为这正是当初写作本书的初衷。

目　录

第一部分　掀起Spring的盖头来

第1章　Spring框架的由来……2

1.1　Spring之崛起……2
1.2　Spring框架概述……3
1.3　Spring大观园……5
1.4　小结……8

第二部分　Spring的IoC容器

第2章　IoC的基本概念……10

2.1　我们的理念是：让别人为你服务……10
2.2　手语，呼喊，还是心有灵犀……13
　2.2.1　构造方法注入……13
　2.2.2　setter方法注入……13
　2.2.3　接口注入……14
　2.2.4　三种注入方式的比较……15
2.3　IoC的附加值……15
2.4　小结……17

第3章　掌管大局的IoC Service Provider……18

3.1　IoC Service Provider的职责……18
3.2　运筹帷幄的秘密——IoC Service Provider如何管理对象间的依赖关系……19
　3.2.1　直接编码方式……19
　3.2.2　配置文件方式……20
　3.2.3　元数据方式……21
3.3　小结……21

第4章　Spring的IoC容器之BeanFactory……22

4.1　拥有BeanFactory之后的生活……24
4.2　BeanFactory的对象注册与依赖绑定方式……26
　4.2.1　直接编码方式……26
　4.2.2　外部配置文件方式……28
　4.2.3　注解方式……31
4.3　BeanFactory的XML之旅……33
　4.3.1　<beans>和<bean>……33
　4.3.2　孤孤单单一个人……35
　4.3.3　Help Me, Help You……36
　4.3.4　继承？我也会！……50
　4.3.5　bean的scope……51
　4.3.6　工厂方法与FactoryBean……56
　4.3.7　偷梁换柱之术……61
4.4　容器背后的秘密……66
　4.4.1　“战略性观望”……66
　4.4.2　插手“容器的启动”……67
　4.4.3　了解bean的一生……74
4.5　小结……85

第5章　Spring IoC容器ApplicationContext……86

5.1　统一资源加载策略……86
　5.1.1　Spring中的Resource……87
　5.1.2　ResourceLoader，“更广义的URL”……88
　5.1.3　ApplicationContext与ResourceLoader……91
5.2　国际化信息支持（I18n MessageSource）……97
　5.2.1　Java SE提供的国际化支持……97
　5.2.2　MessageSource与ApplicationContext……98

5.3 容器内部事件发布 …… 102
5.3.1 自定义事件发布 …… 102
5.3.2 Spring的容器内事件发布类结构分析 …… 105
5.3.3 Spring容器内事件发布的应用 … 107
5.4 多配置模块加载的简化 …… 109
5.5 小结 …… 110

第6章 Spring IoC 容器之扩展篇 …… 111

6.1 Spring 2.5的基于注解的依赖注入 …… 111
6.1.1 注解版的自动绑定（@Autowired）…… 111
6.1.2 @Autowired之外的选择——使用JSR250标注依赖注入关系 …… 115
6.1.3 将革命进行得更彻底一些（class-path-scanning功能介绍）…… 116
6.2 Spring 3.0展望 …… 119
6.3 小结 …… 120

第三部分 Spring AOP 框架

第7章 一起来看 AOP …… 122

7.1 AOP的尴尬 …… 124
7.2 AOP走向现实 …… 125
7.2.1 静态AOP时代 …… 125
7.2.2 动态AOP时代 …… 126
7.3 Java平台上的AOP实现机制 …… 126
7.3.1 动态代理 …… 126
7.3.2 动态字节码增强 …… 126
7.3.3 Java代码生成 …… 127
7.3.4 自定义类加载器 …… 127
7.3.5 AOL扩展 …… 127
7.4 AOP国家的公民 …… 128
7.4.1 Joinpoint …… 128
7.4.2 Pointcut …… 130
7.4.3 Advice …… 131
7.4.4 Aspect …… 133
7.4.5 织入和织入器 …… 133
7.4.6 目标对象 …… 133
7.5 小结 …… 134

第8章 Spring AOP 概述及其实现机制 …… 135

8.1 Spring AOP概述 …… 135
8.2 Spring AOP的实现机制 …… 136
8.2.1 设计模式之代理模式 …… 136
8.2.2 动态代理 …… 139
8.2.3 动态字节码生成 …… 141
8.3 小结 …… 142

第9章 Spring AOP 一世 …… 143

9.1 Spring AOP中的Joinpoint …… 143
9.2 Spring AOP中的Pointcut …… 144
9.2.1 常见的Pointcut …… 146
9.2.2 扩展Pointcut（Customize Pointcut）…… 151
9.2.3 IoC容器中的Pointcut …… 152
9.3 Spring AOP中的Advice …… 153
9.3.1 per-class类型的Advice …… 153
9.3.2 per-instance类型的Advice …… 159
9.4 Spring AOP中的Aspect …… 163
9.4.1 PointcutAdvisor家族 …… 164
9.4.2 IntroductionAdvisor分支 … 167
9.4.3 Ordered的作用 …… 168
9.5 Spring AOP的织入 …… 170
9.5.1 如何与ProxyFactory打交道 … 170
9.5.2 看清ProxyFactory的本质 …… 175
9.5.3 容器中的织入器——ProxyFactoryBean …… 179
9.5.4 加快织入的自动化进程 …… 185
9.6 TargetSource …… 190
9.6.1 可用的TargetSource实现类 … 191
9.6.2 自定义TargetSource …… 195
9.7 小结 …… 197

第10章 Spring AOP 二世 …… 198

10.1 @AspectJ形式的Spring AOP …… 198
10.1.1 @AspectJ形式AOP使用之先睹为快 …… 199
10.1.2 @AspectJ形式的Pointcut …… 201
10.1.3 @AspectJ形式的Advice …… 211
10.1.4 @AspectJ中的Aspect更多话题 …… 220

10.2 基于Schema的AOP……223
10.2.1 基于Schema的AOP配置概览……223
10.2.2 向基于Schema的AOP迁移……225
10.2.3 @AspectJ到"基于Schema的AOP"迁移……227
10.3 小结……235

第 11 章 AOP 应用案例……237
11.1 异常处理……237
11.1.1 Java异常处理……237
11.1.2 Fault Barrier……238
11.2 安全检查……239
11.3 缓存……240
11.4 小结……240

第 12 章 Spring AOP 之扩展篇……241
12.1 有关公开当前调用的代理对象的探讨……241
12.1.1 问题的现象……241
12.1.2 原因的分析……242
12.1.3 解决方案……243
12.2 小结……245

第四部分 使用 Spring 访问数据

第 13 章 统一的数据访问异常层次体系……249
13.1 DAO模式的背景……249
13.2 梦想照进现实……251
13.3 发现问题，解决问题……252
13.4 不重新发明轮子……254
13.5 小结……257

第 14 章 JDBC API 的最佳实践……258
14.1 基于Template的JDBC使用方式……258
14.1.1 JDBC的尴尬……258
14.1.2 JdbcTemplate的诞生……261
14.1.3 JdbcTemplate和它的兄弟们……274
14.1.4 Spring中的DataSource……296
14.1.5 JdbcDaoSupport……301
14.2 基于操作对象的JDBC使用方式……302
14.2.1 基于操作对象的查询……303
14.2.2 基于操作对象的更新……310
14.2.3 基于操作对象的存储过程调用……313
14.3 小结……316

第 15 章 Spring 对各种 ORM 的集成……317
15.1 Spring对Hibernate的集成……318
15.1.1 旧日"冬眠"时光……318
15.1.2 "春天"里的"冬眠"……321
15.2 Spring对iBATIS的集成……329
15.2.1 iBATIS实践之"前生"篇……329
15.2.2 iBATIS实践之"今世"篇……331
15.3 Spring中对其他ORM方案的集成概述……337
15.3.1 Spring对JDO的集成……337
15.3.2 Spring对TopLink的集成……340
15.3.3 Spring对JPA的集成……341
15.4 小结……344

第 16 章 Spring 数据访问之扩展篇……345
16.1 活用模板方法模式及Callback……345
16.1.1 FTPClientTemplate……345
16.1.2 HttpClientTemplate……349
16.2 数据访问中的多数据源……350
16.2.1 "主权独立"的多数据源……350
16.2.2 "合纵连横"的多数据源……352
16.2.3 结束语……354
16.3 Spring 3.0展望……356
16.4 小结……356

第五部分 事务管理

第 17 章 有关事务的楔子……358
17.1 认识事务本身……358
17.2 初识事务家族成员……360
17.3 小结……362

第 18 章 群雄逐鹿下的 Java 事务管理……363
18.1 Java平台的局部事务支持……363
18.2 Java平台的分布式事务支持……365
18.2.1 基于JTA的分布式事务管理……366

18.2.2 基于JCA的分布式事务管理 367
18.3 继续前行之前的反思 367
18.4 小结 369

第 19 章 Spring 事务王国的架构 370

19.1 统一中原的过程 371
19.2 和平年代 376
19.2.1 TransactionDefinition 376
19.2.2 TransactionStatus 382
19.2.3 PlatformTransactionManager 382
19.3 小结 392

第 20 章 使用 Spring 进行事务管理 393

20.1 编程式事务管理 393
20.1.1 直接使用PlatformTransactionManager进行编程式事务管理 393
20.1.2 使用TransactionTemplate进行编程式事务管理 394
20.1.3 编程创建基于Savepoint的嵌套事务 396
20.2 声明式事务管理 397
20.2.1 引子 397
20.2.2 XML元数据驱动的声明式事务 399
20.2.3 注解元数据驱动的声明式事务 410
20.3 小结 413

第 21 章 Spring 事务管理之扩展篇 414

21.1 理解并活用ThreadLocal 414
21.1.1 理解ThreadLocal的存在背景 414
21.1.2 理解ThreadLocal的实现 415
21.1.3 ThreadLocal的应用场景 416
21.1.4 使用ThreadLocal管理多数据源切换的条件 417
21.2 谈Strategy模式在开发过程中的应用 420
21.3 Spring与JTA背后的奥秘 423
21.4 小结 427

第六部分 Spring 的 Web MVC 框架

第 22 章 迈向 Spring MVC 的旅程 430

22.1 Servlet独行天下的时代 430
22.2 繁盛一时的JSP时代 433
22.3 Servlet与JSP的联盟 436
22.4 数英雄人物，还看今朝 438
22.5 小结 440

第 23 章 Spring MVC 初体验 441

23.1 鸟瞰Spring MVC 442
23.2 实践出真知 446
23.2.1 Spring MVC应用的物理结构 447
23.2.2 按部就班地开始工作 451
23.3 小结 459

第 24 章 近距离接触 Spring MVC 主要角色 460

24.1 忙碌的协调人HandlerMapping 460
24.1.1 可用的HandlerMapping 461
24.1.2 HandlerMapping执行序列（Chain Of HandlerMapping） 463
24.2 我们的亲密伙伴Controller 464
24.2.1 AbstractController 465
24.2.2 MultiActionController 468
24.2.3 SimpleFormController 476
24.2.4 AbstractWizardFormController 496
24.2.5 其他可用的Controller实现 503
24.3 ModelAndView 505
24.3.1 ModelAndView中的视图信息 505
24.3.2 ModelAndView中的模型数据 506
24.4 视图定位器ViewResolver 506
24.4.1 可用的ViewResolver实现类 507
24.4.2 ViewResolver查找序列（Chain Of ViewResolver） 511

24.5 各司其职的View ······ 511
24.5.1 View实现原理回顾 ······ 512
24.5.2 可用的View实现类 ······ 515
24.5.3 自定义View实现 ······ 521
24.6 小结 ······ 523

第 25 章 认识更多 Spring MVC 家族成员 ······ 524
25.1 文件上传与MultipartResolver ······ 525
25.1.1 使用MultipartResolver进行文件上传的简单分析 ······ 526
25.1.2 文件上传实践 ······ 527
25.2 Handler与HandlerAdaptor ······ 530
25.2.1 问题的起源 ······ 530
25.2.2 深入了解Handler ······ 531
25.2.3 近看HandlerAdaptor的奥秘 ······ 533
25.2.4 告知Handler与Handler-Adaptor的存在 ······ 535
25.3 框架内处理流程拦截与Handler-Interceptor ······ 536
25.3.1 可用的Handler-Interceptor实现 ······ 537
25.3.2 自定义实现Handler-Interceptor ······ 538
25.3.3 HandlerInterceptor寻根 ······ 540
25.3.4 HandlerInterceptor之外的选择 ······ 541
25.4 框架内的异常处理与Handler-ExceptionResolver ······ 544
25.5 国际化视图与LocalResolver ······ 548
25.5.1 可用的LocaleResolver ······ 549
25.5.2 LocaleResolver的足迹 ······ 550
25.5.3 Locale的变更与LocaleChangeHandler ······ 551
25.6 主题（Theme）与ThemeResolver ······ 552
25.6.1 提供主题资源的ThemeSource ······ 552
25.6.2 管理主题的ThemeResolver ······ 554
25.6.3 切换主题的ThemeChange-Interceptor ······ 555
25.7 小结 ······ 556

第 26 章 Spring MVC 中基于注解的 Controller ······ 557
26.1 初识基于注解的Controller ······ 557
26.2 基于注解的Controller原型分析 ······ 558
26.2.1 自定义用于基于注解的Controller的HandlerMapping ······ 558
26.2.2 自定义用于基于注解的Controller的HandlerAdaptor ······ 560
26.3 近看基于注解的Controller ······ 563
26.3.1 声明基于注解的Controller ······ 563
26.3.2 请求参数到方法参数的绑定 ······ 569
26.3.3 使用@ModelAttribute访问模型数据 ······ 572
26.3.4 通过@SessionAttribute管理Session数据 ······ 574
26.4 小结 ······ 576

第 27 章 Spring MVC 之扩展篇 ······ 577
27.1 Spring MVC也Convention Over Configuration ······ 577
27.1.1 Convention Over Configuration简介 ······ 577
27.1.2 Spring MVC中的Convention Over Configuration ······ 578
27.2 Spring 3.0展望 ······ 581
27.3 小结 ······ 582

第七部分 Spring 框架对 J2EE 服务的集成和支持

第 28 章 Spring 框架内的 JNDI 支持 ······ 584
28.1 JNDI简单回顾 ······ 584
28.2 Spring框架内JNDI访问的基石——JndiTemplate ······ 585
28.3 JNDI对象的依赖注入——JndiObjectFactoryBean ······ 587
28.4 小结 ······ 588

第 29 章 Spring 框架对 JMS 的集成 589
29.1 说说JMS的身世 589
29.2 使用JMS API进行应用开发的传统套路 590
29.3 Spring改进后的JMS实战格斗术 592
29.3.1 消息发送和同步接收 592
29.3.2 异步消息接收 601
29.3.3 JMS相关异常处理 607
29.3.4 框架内的事务管理支持 608
29.4 小结 609
第 30 章 使用 Spring 发送 E-mail 610
30.1 思甜前，先忆苦 610
30.2 Spring的E-mail抽象层分析 612
30.2.1 直接创建邮件消息并发送 614
30.2.2 使用MimeMessage-Preparator发送邮件 615
30.3 Spring的E-mail支持在实际开发中的应用 616
30.4 小结 622
第 31 章 Spring 中的任务调度和线程池支持 623
31.1 Spring与Quartz 623
31.1.1 初识Quartz 623
31.1.2 融入Spring大家庭的Quartz 626
31.2 Spring对JDK Timer的集成 631
31.2.1 JDK Timer小记 631
31.2.2 Spring集成后的JDK Timer 632
31.3 Executor的孪生兄弟TaskExecutor 634
31.3.1 可用的TaskExecutor 635
31.3.2 TaskExecutor使用实例 637
31.4 小结 639
第 32 章 Spring 框架对 J2EE 服务的集成之扩展篇 640
32.1 MailMonitor的延伸 640
32.2 Spring 3.0展望 642
32.3 小结 642
第 33 章 Spring 远程方案 643
33.1 从“对面交谈”到“千里传声” 643
33.2 Spring Remoting架构分析 645
33.2.1 Spring Remoting之远程访问异常体系 645
33.2.2 统一风格的远程服务公开与访问方式 646
33.3 Spring Remoting提供的远程服务支持 648
33.3.1 基于RMI的Remoting方案 648
33.3.2 基于HTTP的轻量级Remoting方案 651
33.3.3 基于Web服务的远程方案 655
33.3.4 基于JMS的远程方案 658
33.4 扩展Spring Remoting 660
33.5 Spring Remoting之扩展篇 663
33.5.1 拉开JMX演出的序幕 663
33.5.2 Spring 3.0展望 664
参考文献 665

Part 1 第一部分

掀起 Spring 的盖头来

本部分内容

■ 第 1 章　Spring 框架的由来

第1章 Spring框架的由来

1

本章内容

- Spring之崛起
- Spring框架概述
- Spring大观园

J2EE作为一种企业级应用开发平台，其优异表现是我们有目共睹的。但纵使是最为强大的军队，如果没有一个好的指挥官，不知道如何发挥这支军队的强大战斗力，那这支军队也不可能取得太多辉煌的战果。在J2EE平台的一些早期实践中，就出现了对J2EE平台所提供的各项服务的滥用，将基于J2EE平台的企业级开发带入了窘境。

Spring是于2003年兴起的一个轻量级的Java开发框架，由Rod Johnson在其著作*Expert One-On-One J2EE Development and Design*中阐述的部分理念和原型衍生而来。它的最初目的主要是为了简化Java EE的企业级应用开发，相对于过去EJB[①]时代重量级的企业应用开发而言，Spring框架的出现为曾经阴霾的天空带来了灿烂的阳光。

1.1 Spring之崛起

在中世纪的欧洲，当重装骑兵所向披靡时，哪国的军队中如果没有一支重装骑兵真的会让人笑话的，按照电影《大腕》里的一句话说“你都不好意思跟人打招呼”。应该说，在当时的历史/军事环境下，重装骑兵在军队中确实发挥了不可或缺的作用。有时候，一次关键时刻的重装骑兵冲锋就可以奠定战局的胜利。但是，时过境迁，历史的车轮一直在向前缓缓行进，重装骑兵头上的光环也随之渐趋黯淡，其缺点开始显露无遗。

- 重装骑兵代价高昂。一名重装骑兵的装备花费几乎能够武装一小队轻步兵，对于财力不够雄厚的国家来说，维持一支常备的重装骑兵队伍绝非易事。实际上，对于财力雄厚的大国（相当于IT界的IBM、微软）来说，为了减轻财政上的压力，通常也是将这部分花销尽量摊派给贵族。
- 兵种自身限制太多。沉重的盔甲以及一整套装备使得重装骑兵的机动性和灵活性大打折扣，在正式投入战斗之前，重装骑兵需要很长时间的列装和部署，对于瞬息万变的战场形势来说，某些情况下，这等同于自杀。
- 发挥作用的场景有限。纵使各翼军队能够掩护重装骑兵完成部署，但如果战场地形不适合重装骑兵冲锋，那也就无法让他们大显身手，前期的准备或者战斗掩护就更是得不偿失。

当新的战术和兵种能够更加高效地完成作战任务时，依然主打重装骑兵的军队能取得什么样的战

① 此后提到的EJB通常都是指1.*x*和2.*x*版本的EJB，因为EJB3现在已经受Spring框架的影响，也主打基于POJO的轻量级应用解决方案了。

果，轻骑兵的出现已经给了我们一个明确的答案。

当然，我的本意不在讲战争史，我只是想让大家看到，在早期的J2EE平台开发实践过程中，盲目地推崇某一“兵种”，比如EJB，将会招致怎么样的一种命运。历史是相似的，不管是军事发展史，还是我们的J2EE平台企业级应用开发史。君不见当年言必称EJB的盛况吗？这跟中世纪欧洲重装骑兵的显赫是何等地相似。但任何技术或者事物都有其适用的场景，如果用在了不合适的场景下，我们就不得不为滥用而付出相应的代价。EJB是使用J2EE平台各项服务的一种方式，但不是唯一的方式。对于分布式系统来说，使用EJB在某些方面确实可以带给我们很大的收益，但并不是所有的J2EE应用都要用于分布式环境。如果不分场景地滥用EJB，J2EE平台上的这支“重装骑兵”的局限性自然会暴露出来。

- 使用EJB，通常也就意味着需要引入拥有EJB Container的应用服务器（J2EE Application Server）的支持，比如BEA的WebLogic或者IBM的WebSphere，而这往往也就意味着高昂的授权费用。虽然开源的JBoss等应用服务器也提供对EJB Container的支持，但对于商业软件来说，出于产品质量和售后服务等因素考虑，商业产品WebLogic或者WebSphere依然是最佳选择。这跟当年欧洲崇尚重装骑兵相似，都免不了要付出高昂的代价。
- 说到重装骑兵列装和部署的复杂和迟缓，使用EJB开发的企业级应用也存在同样的问题。使用EJB使得应用程序的部署和测试都更加困难，复杂的类加载机制、复杂的部署描述符、过长的开发部署周期等，无疑都增加了开发EJB应用程序的难度。
- 重装骑兵需要合适的战机才能发挥其最大作用，EJB也是如此。只有在分布式的场景中，EJB才会带给我们最大的益处。但是，当大家在崇拜EJB的狂热氛围之下，不分场景地滥用它时，后果自然可想而知了。要么是开发过程缓慢，无法如期交付；要么就是程序交付后性能极差，很难满足客户需求。

当然我们还可以列举更多，但这些已经可以说明问题了。应该说，基于EJB的应用开发实践并非一无是处。尽管其本身在合适的场景下也或多或少地存在一些问题，但不分场景地滥用它，才会导致基于EJB的应用开发声名狼藉。历史的车轮时刻都没有停下，当人们意识到大部分J2EE应用开发初期甚至整个生命周期内都不需要牵扯到分布式架构时，J2EE平台上对应EJB而生的“轻骑兵”自然也该千呼万唤始出来了——倡导轻量级应用解决方案的Spring，就承担了这一历史角色！

Spring倡导一切从实际出发，以实用的态度来选择适合当前开发场景的解决方案。如果不需要用到分布式架构，那就没有必要使用EJB之类的“牛刀”。而大多数的J2EE应用也确实不需要在开发初期或者整个生命周期内引入任何分布式架构。这个时候，采用敏捷、轻量级的开发方案可以收到更好的效果。Spring所倡导的J2EE轻量级应用解决方案顺应天时，自然得以快速崛起……

注意　没有任何一种解决方案是普遍适用的，只有适用于特定场景的解决方案，脱离具体场景来讨论任何解决方案都是脱离实际的表现。Spring并不是要替代EJB，而是给出EJB之外的另一种方案而已，甚至于二者可以是互补的。如果某一场景下EJB依然是最为合适的解决方案，那么我们可以毫不迟疑地使用它；同样地，对于Spring的使用也应该考虑到具体的应用场景，这一点应该是需要我们始终牢记的。当古代的重装骑兵以坦克的形式重新跃上历史舞台时，Java平台上会不会也上演同样的一幕呢？我们拭目以待。

1.2　Spring 框架概述

大气层的包裹为地球创造了适合人类生存的环境。当地球上的资源足够满足人类生存需要时，我

们没必要急于开展星际移民计划，那样造成的人力物力开销会很大。当年盲目倡导EJB架构，就好像从一开始就不顾实际情况而开展星际移民计划，向其他星球移民当然是个好想法，但是否立即有必要如此大动干戈呢？对于大多数的应用程序来说，EJB架构是没有必要的。我们只需要一个大气层包裹的、环境适宜的地球即可。Spring框架所倡导的基于POJO（Plain Old Java Object，简单Java对象）的轻量级开发理念，就是从实际出发，立足于最基础的POJO（就好像我们的地球）。为了能够让这些基础的POJO构建出健壮而强大的应用，Spring框架就好像那包裹地球的大气层一样，为构筑应用的POJO提供了各种服务，进而创造了一套适宜用POJO进行轻量级开发的环境。

从广义上讲，不管Spring框架自发布到现在经过了多少次的版本更迭（从1.*x*到2.0再到2.5），其本质是始终不变的，都是为了提供各种服务，以帮助我们简化基于POJO的Java应用程序开发。Spring框架为POJO提供的各种服务共同组成了Spring的生命之树，如图1-1所示。

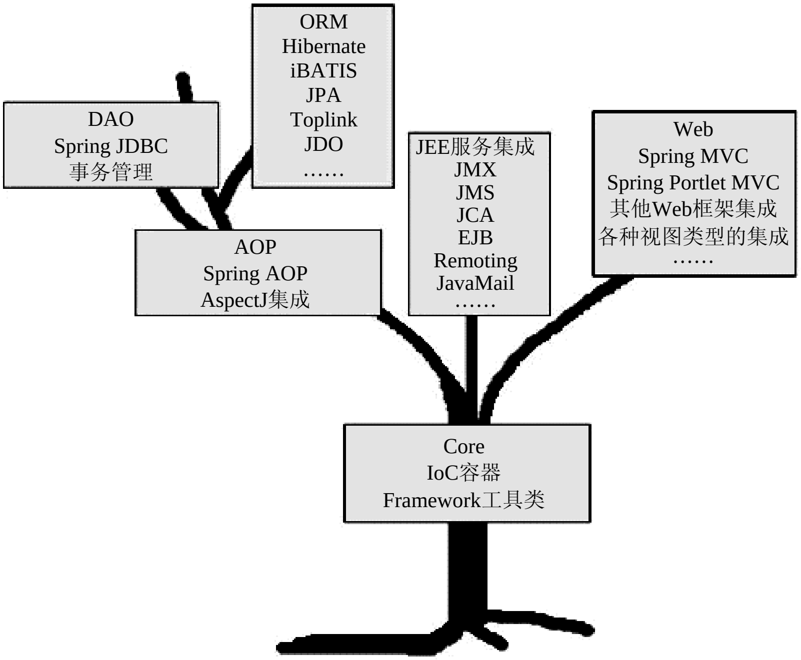

图1-1　Spring框架总体结构

组成整个Spring框架的各种服务实现被划分到了多个相互独立却又相互依赖的模块当中。正如我们在图1-1中所见到的那样，这些模块组成了Spring生命之树的枝和干，说白了也就是它们组成了Spring框架的核心骨架。抓住了这副骨架，也就抓住了Spring框架的学习主线。

整个Spring框架构建在Core核心模块之上，它是整个框架的基础。在该模块中，Spring为我们提供了一个IoC容器（IoC Container）实现，用于帮助我们以依赖注入的方式管理对象之间的依赖关系。对Spring的IoC容器的介绍将成为我们此次Spring之旅的第一站。除此之外，Core核心模块中还包括框架内部使用的各种工具类（如果愿意，我们也可以在框架之外使用），比如Spring的基础IO工具类等，这些基础工具类我们也会在合适的地方介绍。

沿着Spring生命之树往上左序遍历，我们将来到AOP模块。该模块提供了一个轻便但功能强大的AOP框架，让我们可以以AOP的形式增强各POJO的能力，进而补足OOP/OOSD之缺憾。Spring的AOP框架符合AOP Alliance规范，采用Proxy模式构建，与IoC容器相结合，可以充分显示出Spring AOP的强大威力。我们将在了解了Spring的IoC容器的基础上，详细讲述Spring AOP这一部分。

继续上行，Spring框架在Core核心模块和AOP模块的基础上，为我们提供了完备的数据访问和事务管理的抽象和集成服务。在数据访问支持方面，Spring对JDBC API的最佳实践极大地简化了该API的使用。除此之外，Spring框架为各种当前业界流行的ORM产品，比如Hibernate、iBATIS、Toplink、JPA等提供了形式统一的集成支持。Spring框架中的事务管理抽象层是Spring AOP的最佳实践，它直接构建在Spring AOP的基础之上，为我们提供了编程式事务管理和声明式事务管理的完备支持。这些服务极大地简化了日常应用开发过程中的数据访问和事务管理工作。在学习完这两部分内容之后，相信读者将会有切身的体会。

为了简化各种Java EE服务（像JNDI、JMS以及JavaMail等）的使用，Spring框架为我们提供了针对这些Java EE服务的集成服务。在Spring的帮助下，这些Java EE服务现在都变得不再烦琐难用。因为相关的Java EE服务较多，我们将会选择合适的几种介绍Spring框架给予它们的支持。随着航空航天技术的发展，我们现在可以从地球上发送飞船去访问其他星球，使用Spring框架构建的基于POJO的应用程序如果也需要远程访问或者公开一些服务的话，Spring的Remoting框架将帮助它完成这一使命。Spring的Remoting框架和Spring对其他Java EE服务的集成将分别在不同的章节中介绍。

最后要提到的就是Web模块。在该模块中，Spring框架提供了一套自己的Web MVC框架，职责分明的角色划分让这套框架看起来十分地“醒目”。我们将为Spring的Web MVC框架单独开辟一块“领地”进行讲解。在那一部分中，读者可以充分领略Web MVC框架的魅力。Spring的Portlet MVC构建在Spring Web MVC之上，延续了Spring Web MVC的一贯风格。本书不会对其做详细介绍，如果需要，可以参考文献中的有关参考书籍。Spring Web MVC并不排斥现有的其他Web框架，像Struts、WebWork以及JSF等；Spring的Web框架都为它们提供了集成支持。除此之外，像Web开发中可能牵扯的各种视图（View）技术，Spring Web框架更是给予了足够的重视。

就像一棵树必须依赖强大的根基才能生长繁盛一样，Spring框架内的各个模块也是如此。理论上来说，上层的模块需要依赖下层的模块才能正常工作，这就是为什么说这些模块是相互依赖的。不过，近乎处于同一水平线的各个模块之间却可以认为是相互独立的，彼此之间没什么瓜葛。从这个角度看，这些模块之间的相互独立一说也是成立的。

以上就是对整个Spring框架的总体介绍。在开始愉快的Spring旅程之前，我想带大家先逛一逛“Spring大观园”，这样，大家就会发现即将开始的Spring之旅更加值得期待。

注意 不要只将Spring看作是一个IoC容器，也不要只将Spring与AOP挂钩，Spring提供的远比这些东西要多得多。Spring不仅仅是一个简化Java EE开发的轻量级框架，它更应该是一个简化任何Java应用的开发框架。如果你愿意，甚至可以在Java的三个平台上（J2SE、J2EE、J2ME）应用Spring框架。即使当前的Spring框架还不支持相应平台或者相应场景的应用开发，但是只要你掌握了Spring的理念和方法，同样可以让新的“Spring”在相应的场景中发挥作用。

1.3 Spring 大观园

在1995年Java作为一门计算机语言而诞生时，有谁能够想到，短短10多年间，它已经发展成为一

个强大的开发平台？对于Spring框架来说，历史又在重演，而且几乎毫无悬念。

Spring大观园中有一棵参天大树，它得以茁壮成长，主要因为它有一个好的根基，那就是Spring框架。在Spring框架的基础上，Spring家族人丁开始兴旺，不断涌现出一个又一个引人注目的家族成员，包括：

- Spring Web Flow（SWF）[①]。Spring Web Flow构建于Spring Web MVC框架之上，旨在简化拥有复杂用户交互逻辑的Web应用程序的开发。通过Spring Web Flow的扩展支持，可以在基于Spring Web MVC的Web应用程序中以更简单的方式，创建更加复杂的业务交互流程。同时，Spring Web Flow还让Ajax和JSF享受一等公民待遇，所有这些将帮助我们更快更好地满足各种用户的实际需求。
- Spring Web Services[②]。Spring Web Services是一套采用契约优先（Contract-First）开发模式，创建文档驱动（Document-driven）Web服务的Web服务开发框架。它除了对Web服务中涉及的XML的映射关系管理提供了详尽的支持，还与Spring框架以及其他子项目（比如Spring Security）紧密结合，帮助以更加灵活高效的方式打造Web服务应用服务。
- Spring Security（原来的Acegi Security）[③]。Spring Security由原来的Acegi Security发展而来，主要为基于Spring框架的企业级应用程序提供安全解决方案。Spring Security 2.0发布后在原来Acegi Security 1.0的基础上又添加了很多富有吸引力的特性，包括简化配置、面向RESTful请求的安全认证、与Spring Web Flow和Spring Web Services等项目的良好集成等，可以说为基于Spring框架的企业级应用提供了一站式的安全方面的解决方案。
- Spring Dynamic Modules for OSGi Service Platforms[④]。Spring-DM是融合了Spring框架以及OSGi两家优良基因后的产物，它集Spring框架各种服务和OSGi的动态性、模块化等特性于一身，可以帮助我们以一种全新的方式来打造新一代的企业级应用程序。SpringSource Application Platform应用服务器就是构建在Spring-DM之上的。在企业级应用开发领域，Spring-DM或许会掀起另一个浪潮。
- Spring Batch[⑤]。当意识到企业应用中批处理业务所占的市场份额不容小觑之后，Spring Batch开始浮出水面，它是构建在Spring框架之上的一套轻量级批处理开发框架，由SpringSource和埃森哲（Accenture）合力打造。如果你还在为无法找到一款满意的开源批处理开发框架而烦恼，也许Spring Batch会让你的烦恼顷刻间烟消云散。
- Spring Integration[⑥]。Spring Integration面向创建基于Spring开发框架的企业集成（Enterprise Integration）解决方案，对*Enterprise Integration Patterns*[⑦]一书中的企业集成模式提供支持。它在现有Spring框架对企业方案集成的基础上，提出了更高层次的抽象方案，使得业务和集成逻辑得以松散耦合，很好地分离了企业集成过程中的不同关注点。
- Spring LDAP[⑧]。Spring LDAP传承了Spring框架中应用模板方法模式（Template Method Pattern）

---

① http://www.springframework.org/webflow。
② http://www.springframework.org/spring-ws。
③ http://static.springframework.org/spring-security/site/index.html。
④ http://www.springframework.org/osgi。
⑤ http://www.springframework.org/spring-batch。
⑥ http://www.springframework.org/spring-integration。
⑦ http://www.eaipatterns.com/。
⑧ http://www.springframework.org/ldap。

的优良传统，由最初的LdapTemplate发展演化而来，旨在简化LDAP相关操作。

- Spring IDE[①]。如果读者使用Eclipse平台开发Spring应用程序，结合Spring IDE插件将会使开发更加得心应手。Spring IDE以Eclipse开发平台为中心，想开发人员之所想，包含了各种实用的特性，为使用Eclipse创建基于Spring的应用程序，提供了灵活而强大的开发环境。
- Spring Modules[②]。为了避免Spring框架对各种其他项目的集成和支持造成Spring框架本身的臃肿等一系列问题，Spring Modules将那些可选的工具和附加类库剥离出Spring核心框架，纳入自身进行统一管理。如果在使用Spring框架开发的过程中，发现某些第三方库或工具，在核心框架中不存在的话，可以求助于Spring Modules提供的各种扩展，包括它对ANT、OSWorkflow、Apache OJB，以及低版本的iBatis等第三方库的扩展支持。
- Spring JavaConfig[③]。Spring框架提供的依赖注入支持，最初是使用XML表述依赖注入关系的。在Spring 2.5正式提供了基于注解的依赖注入方式之前，Spring JavaConfig就为Spring框架提出了一套基于注解的依赖注入解决方案，它可以看作是Spring 2.5中基于注解的依赖注入正式方案之外的另一种选择。
- Spring Rich Client[④]。与Eclipse RCP为基于SWT/JFace的GUI应用提供了一套完备的开发框架类似，Spring也为使用Swing进行GUI开发的应用提供了一套开发框架，这就是Spring Rich Client。如果你想在开发Swing应用的过程中同时获得Spring框架的各项支持的话，那Spring Rich Client正是为你而生的。
- Spring .NET[⑤]。Spring框架在Java平台上的成功是有目共睹的，这种成功同样渗透到了.NET平台，Spring .NET就是SpringSource为.NET企业开发平台量身打造的开源应用开发框架。
- Spring BeanDoc[⑥]。Spring BeanDoc可以根据Spring应用程序使用的配置文件中的相应信息，创建对应的文档和图表，帮助我们以更加直观的方式来了解Spring应用程序的整体结构。

这些家族成员全部以Apache Lisence Version 2.0协议发布，共同组成了Spring Projects组合。这一组合对软件开发中的各种需求提供从广度到深度的支持，极大地简化了日常开发工作。而且，因为它们都是以开源形式发布的，所以大部分软件公司都可以从中受益，可以投入更低的成本打造高质量的软件产品。

Spring Projects组合中的开源项目为很多软件开发人员、软件公司带来了益处，但因为是开源，所以这些项目主要靠社区来推动和发展。活跃的开发社区可以为我们带来快速的反馈和支持，但没有任何主体或个人可以保证我们所需要的反馈和支持能够及时有效地得到满足。鉴于这一点，SpringSource（原来的Interface21，即Spring框架的“东家”）在Spring Projects组合的基础上，提供了Spring Portfolio[⑦]产品，SpringSource为Spring Portfolio产品中的各成员提供咨询、培训和支持服务。Spring Portfolio产品由Spring Projects组合中多个成功的企业级开源产品，以及AspectJ等Spring组织外部的优秀开源产品共同组成：

- Spring Framework；

① http://springide.org/。
② https://springmodules.dev.Java.net/。
③ http://www.springframework.org/Javaconfig。
④ http://www.springframework.org/spring-rcp。
⑤ http://www.springframework.net/。
⑥ http://spring-beandoc.sourceforge.net/。
⑦ Portfolio指的是证券投资组合，在这里应该是多种Spring家族相关产品的组合。

- Spring Security（原来的Acegi）；
- Spring Web Flow；
- Spring Web Services；
- Spring Dynamic Modules for the OSGi Service Platform；
- Spring Batch；
- Pitchfork（http://www.springsource.com/pitchfork）；
- AspectJ（http://www.eclipse.org/aspectj/）；
- Spring IDE；
- Spring .NET；
- Spring LDAP；
- Spring Rich Client；
- Spring Integration。

实际上，Spring Portfolio只是SpringSource产品组合之一。为了能够为客户提供更多价值，也为了继续保持Spring家族在Java平台企业级开发中一贯的领先地位，SpringSource积极扩展，为我们带来了更多的企业级开发产品，像构建于Spring Dynamic Modules之上的新一代企业级Java应用服务器SpringSource Application Platform，为企业用户量身打造的SpringSource Enterprise等产品。而且，SpringSource还积极吸纳其他成功开源产品（包括Apache HTTPD[①]、Apache Tomcat[②]、Apache ActiveMQ[③]等）的主要开发人员，以便更好地为这些开源产品的客户提供及时有效的支持服务。

无论从哪一个角度看，整个Spring家族都是富有活力、积极进取的，一旦有新的开发理念或者最佳实践涌现，我们通常会第一时间在Spring家族中发现它们的身影。随着整个Spring平台的发展，我们会看到Spring大观园将愈发地花团锦簇、欣欣向荣。看到这么一幅宏大而美丽的景象，你或许早就热血沸腾，想要马上投入Spring大观园之中。不过，正像Donald J. Trump在*How To Get Rich*一书中所说的那样："Before the dream lifts you into the clouds, make sure look hard at the facts on the ground."[④]

要知道，所有的Spring家族成员全部构建于Spring框架基础之上，在我们想要往Spring这棵参天大树更高的地方攀爬之前，实实在在地先去了解Spring框架这一根基，才是当前的首要任务，不是吗？如果读者能够认同这些，那么现在就让我们来开始Spring框架的全景之旅吧！

1.4　小结

本章首先对Spring框架得以迅速崛起的背景做了简短介绍，然后带领读者从总体上了解了Spring框架的构成。按常规思路，在了解了Spring框架的总体结构之后，就应该深入框架内部以进一步了解它的详细内容。但是，为了让大家对Spring框架以及构建在该框架基础之上的整个平台有所认识，本章最后加入了对整个Spring开发平台的总体介绍。

---

① http://www.covalent.net/supportservices/apache/index.html。
② http://www.covalent.net/supportservices/tomcat/index.html。
③ http://www.covalent.net/supportservices/activemq/index.html。
④ 中文大意是："在被梦想搞得飘飘然之前，最好先让自己脚踏实地"。——编者注

第2章

IoC的基本概念

2

本章内容

- 我们的理念是：让别人为你服务
- 手语，呼喊，还是心有灵犀
- IoC的附加值

2.1 我们的理念是：让别人为你服务

IoC是随着近年来轻量级容器（Lightweight Container）的兴起而逐渐被很多人提起的一个名词，它的全称为Inversion of Control，中文通常翻译为“控制反转”，它还有一个别名叫做依赖注入（Dependency Injection）。好莱坞原则“Don’t call us, we will call you.”[①]恰如其分地表达了“反转”的意味，是用来形容IoC最多的一句话。那么，为什么需要IoC？IoC的具体意义是什么？它到底有什么独到之处？让我们带着这些疑问开始我们的IoC之旅吧。

> **注意** 本章更多的是将IoC和依赖注入看作等同的概念进行讲解。但是，在这一点上可能存在不同的观点，比如*Expert Spring MVC and Web Flow*和*Expert One-on-One J2EE without EJB*等书中都将依赖注入看作是IoC的一种方式。不过，本章暂且忽略这些观点，将IoC和依赖注入等同看待。在读者理解了依赖注入之后，可以再结合其他资料对IoC做进一步的研究。

为了更好地阐述IoC模式的概念，我们引入以下简单场景。

在我经历的FX项目[②]中，经常需要近乎实时地为客户提供外汇新闻。通常情况下，都是先从不同的新闻社订阅新闻来源，然后通过批处理程序定时地到指定的新闻服务器抓取最新的外汇新闻，接着将这些新闻存入本地数据库，最后在FX系统的前台界面显示。

假设我们有一个`FXNewsProvider`类来做以上工作，其代码如代码清单2-1所示。

代码清单2-1 `FXNewsProvider`类的实现

```
public class FXNewsProvider
{
```

① 中文大意是：“你不用找我们，我们会找你的。”——编者注

② FX全称为“Foreign Exchange”，即外汇交易。FX系统通常作为交易的中间商，与上游服务商（比如花旗和莱曼等各大银行）合作，为顾客提供保证金交易服务，顾客只需要交纳很少的保证金就可以进行大额的外汇交易。保证金交易的杠杆可以让顾客“以小博大”，只要很少的资金，顾客就可以享受到汇率变动所带来的收益（当然，评估方向错误也可能招致损失）。

```
    private IFXNewsListener  newsListener;
    private IFXNewsPersister newPersistener;
    public void getAndPersistNews()
    {
      String[] newsIds = newsListener.getAvailableNewsIds();
      if(ArrayUtils.isEmpty(newsIds))
      {
        return;
      }

      for(String newsId : newsIds)
      {
        FXNewsBean newsBean = newsListener.getNewsByPK(newsId);
        newPersistener.persistNews(newsBean);
        newsListener.postProcessIfNecessary(newsId);
      }
    }
}
```

其中，FXNewsProvider需要依赖IFXNewsListener来帮助抓取新闻内容，并依赖IFXNews-Persister存储抓取的新闻。

假设默认使用道琼斯（Dow Jones）新闻社的新闻，那么我们相应地提供了DowJonesNewsLis-tener和DowJonesNewsPersister两个实现。通常情况下，需要在构造函数中构造IFXNewsProvider依赖的这两个类（以下将这种被其他类依赖的类或对象，简称为“依赖类”、“依赖对象”），如代码清单2-2所示。

代码清单2-2 构造IFXNewsProvider类的依赖类

```
public FXNewsProvider()
{
    newsListener   = new DowJonesNewsListener();
    newPersistener = new DowJonesNewsPersister();
}
```

看，这就是我们通常的做事方式！如果我们依赖于某个类或服务，最简单而有效的方式就是直接在类的构造函数中新建相应的依赖类。这就好比要装修新房，需要用家具，这个时候，根据通常解决对象依赖关系的做法，我们就会直接打造出需要的家具来。不过，通常都是分工明确的，所以，大多数情况下，我们可以去家具广场将家具买回来，然后根据需要装修布置即可。

不管是直接打造家具（通过new构造对象），还是去家具广场买家具（或许是通过Service-Locator[①]解决直接的依赖耦合），有一个共同点需要我们关注，那就是，我们都是自己主动地去获取依赖的对象！

可是回头想想，我们自己每次用到什么依赖对象都要主动地去获取，这是否真的必要？我们最终所要做的，其实就是直接调用依赖对象所提供的某项服务而已。只要用到这个依赖对象的时候，它能够准备就绪，我们完全可以不管这个对象是自己找来的还是别人送过来的。对于FXNewsProvider来说，那就是在getAndPersistNews()方法调用newsListener的相应方法时，newsListener能够准备就绪就可以了。如果有人能够在我们需要时将某个依赖对象送过来，为什么还要大费周折地自己去折腾？

实际上，IoC就是为了帮助我们避免之前的“大费周折”，而提供了更加轻松简洁的方式。它的反转，就反转在让你从原来的事必躬亲，转变为现在的享受服务。你想啊，原来还得鞍马劳顿，什么

① J2EE核心模式的一种，主要通过引入中间代理者消除对象间复杂的耦合关系，并统一管理分散的复杂耦合关系。

东西都得自己去拿。现在是用什么，让别人直接送过来就成。所以，简单点儿说，IoC的理念就是，让别人为你服务！在图2-1中，也就是让IoC Service Provider来为你服务！

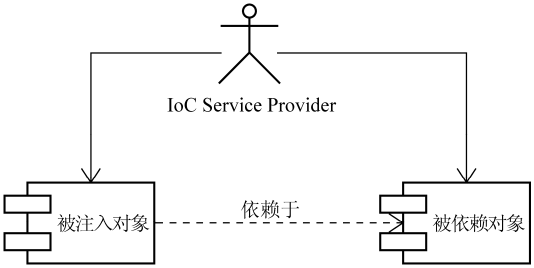

图2-1　IoC的角色

通常情况下，被注入对象会直接依赖于被依赖对象。但是，在IoC的场景中，二者之间通过IoC Service Provider来打交道，所有的被注入对象和依赖对象现在由IoC Service Provider统一管理。被注入对象需要什么，直接跟IoC Service Provider招呼一声，后者就会把相应的被依赖对象注入到被注入对象中，从而达到IoC Service Provider为被注入对象服务的目的。IoC Service Provider在这里就是通常的IoC容器所充当的角色。从被注入对象的角度看，与之前直接寻求依赖对象相比，依赖对象的取得方式发生了反转，控制也从被注入对象转到了IoC Service Provider那里①。

其实IoC就这么简单！原来是需要什么东西自己去拿，现在是需要什么东西就让别人送过来。图2-2以两种场景，形象地说明了使用IoC模式前后的差别。

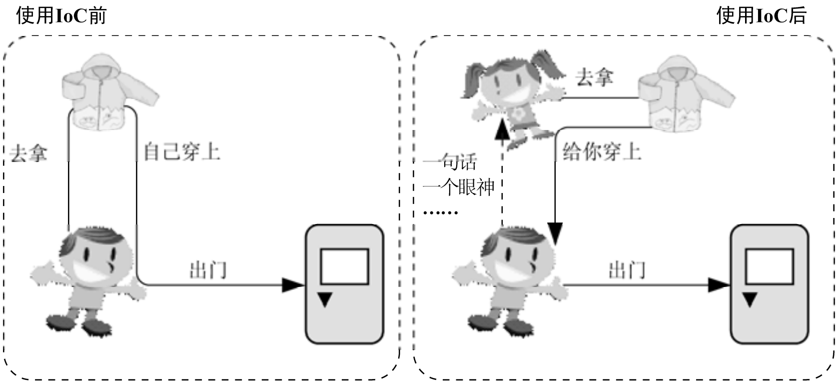

图2-2　使用IoC前后的差别

出门之前得先穿件外套吧？以前，你得自己跑到衣柜前面取出衣服这一依赖对象，然后自己穿上再出门。而现在，你只要跟你的“另一半”使个眼色或说一句“Honey，衣服拿来。”她就会心领神会地到衣柜那里为你取出衣服，然后再给你穿上。现在，你就可以出门了。（此时此刻，你心里肯定窃喜，“有人照顾的感觉真好！”）对你来说，到底哪种场景比较惬意，我想已经不言自明了吧？

① 与之前其他书籍和文章讲解IoC的概念方式不同，本书这里不是从对象解耦的角度来阐述的。为了能让读者将IoC与原来的对象绑定模式做一个对比，我们决定从对象绑定方式的角度来阐述IoC的概念，这样对比可以更加鲜明地表现新概念与老概念的差别。

2.2 手语，呼喊，还是心有灵犀

“伙计，来杯啤酒！”当你来到酒吧，想要喝杯啤酒的时候，通常会直接招呼服务生，让他为你送来一杯清凉解渴的啤酒。同样地，作为被注入对象，要想让IoC Service Provider为其提供服务，并将所需要的被依赖对象送过来，也需要通过某种方式通知对方。

- 如果你是酒吧的常客，或许你刚坐好，服务生已经将你最常喝的啤酒放到了你面前；
- 如果你是初次或偶尔光顾，也许你坐下之后还要招呼服务生，“Waiter,Tsingdao, please.”；
- 还有一种可能，你根本就不知道哪个牌子是哪个牌子，这时，你只能打手势或干脆画出商标图来告诉服务生你到底想要什么了吧！

不管怎样，你终究会找到一种方式来向服务生表达你的需求，以便他为你提供适当的服务。那么，在IoC模式中，被注入对象又是通过哪些方式来通知IoC Service Provider为其提供适当服务的呢？

IoC模式最权威的总结和解释，应该是Martin Fowler的那篇文章“Inversion of Control Containers and the Dependency Injection pattern”，其中提到了三种依赖注入的方式，即构造方法注入（constructor injection）、setter方法注入（setter injection）以及接口注入（interface injection）。下面让我们详细看一下这三种方式的特点及其相互之间的差别。

2.2.1 构造方法注入

顾名思义，构造方法注入，就是被注入对象可以通过在其构造方法中声明依赖对象的参数列表，让外部（通常是IoC容器）知道它需要哪些依赖对象。对于前面例子中的`FXNewsProvider`来说，只要声明如下构造方法（见代码清单2-3）即可支持构造方法注入。

代码清单2-3 `FXNewsProvider`构造方法定义

```
public FXNewsProvider(IFXNewsListener newsListner,IFXNewsPersister newsPersister)
{
   this.newsListener   = newsListner;
   this.newPersistener = newsPersister;
}
```

IoC Service Provider会检查被注入对象的构造方法，取得它所需要的依赖对象列表，进而为其注入相应的对象。同一个对象是不可能被构造两次的，因此，被注入对象的构造乃至其整个生命周期，应该是由IoC Service Provider来管理的。

构造方法注入方式比较直观，对象被构造完成后，即进入就绪状态，可以马上使用。这就好比你刚进酒吧的门，服务生已经将你喜欢的啤酒摆上了桌面一样。坐下就可马上享受一份清凉与惬意。

2.2.2 setter 方法注入

对于JavaBean对象来说，通常会通过`setXXX()`和`getXXX()`方法来访问对应属性。这些`setXXX()`方法统称为setter方法，`getXXX()`当然就称为getter方法。通过setter方法，可以更改相应的对象属性，通过getter方法，可以获得相应属性的状态。所以，当前对象只要为其依赖对象所对应的属性添加setter方法，就可以通过setter方法将相应的依赖对象设置到被注入对象中。以`FXNewsProvider`为例，添加setter方法后如代码清单2-4所示。

代码清单2-4 添加了setter方法声明的`FXNewsProvider`

```
public class FXNewsProvider
```

```
{
  private IFXNewsListener  newsListener;
  private IFXNewsPersister newPersistener;

  public IFXNewsListener getNewsListener() {
    return newsListener;
  }
  public void setNewsListener(IFXNewsListener newsListener) {
    this.newsListener = newsListener;
  }
  public IFXNewsPersister getNewPersistener() {
    return newPersistener;
  }
  public void setNewPersistener(IFXNewsPersister newPersistener) {
    this.newPersistener = newPersistener;
  }
}
```

这样，外界就可以通过调用`setNewsListener`和`setNewPersistener`方法为`FXNewsProvider`对象注入所依赖的对象了。

setter方法注入虽不像构造方法注入那样，让对象构造完成后即可使用，但相对来说更宽松一些，可以在对象构造完成后再注入。这就好比你可以到酒吧坐下后再决定要点什么啤酒，可以要百威，也可以要大雪，随意性比较强。如果你不急着喝，这种方式当然是最适合你的。

2.2.3　接口注入

相对于前两种注入方式来说，接口注入没有那么简单明了。被注入对象如果想要IoC Service Provider为其注入依赖对象，就必须实现某个接口。这个接口提供一个方法，用来为其注入依赖对象。IoC Service Provider最终通过这些接口来了解应该为被注入对象注入什么依赖对象。图2-3演示了如何使用接口注入为`FXNewsProvider`注入依赖对象。

`FXNewsProvider`为了让IoC Service Provider为其注入所依赖的`IFXNewsListener`，首先需要实现`IFXNewsListenerCallable`接口，这个接口会声明一个`injectNewsListner`方法（方法名随意），该方法的参数，就是所依赖对象的类型。这样，`InjectionServiceContainer`对象，即对应的IoC Service Provider就可以通过这个接口方法将依赖对象注入到被注入对象`FXNewsProvider`当中。

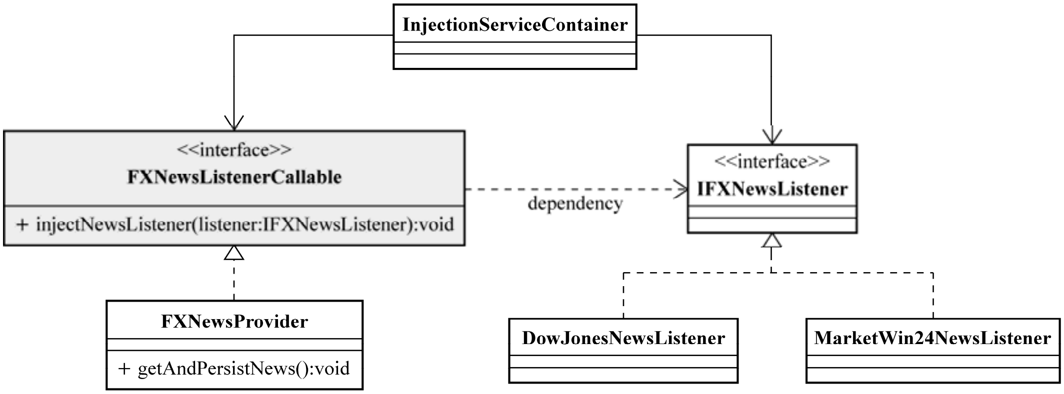

图2-3　使用接口注入的`FXNewsProvider`

> **小心** 在这种情况下，实现的接口和接口中声明的方法名称都不重要。重要的是接口中声明方法的参数类型，必须是“被注入对象”所依赖对象的类型。

接口注入方式最早并且使用最多的是在一个叫做Avalon的项目中，相对于前两种依赖注入方式，接口注入比较死板和烦琐。如果需要注入依赖对象，被注入对象就必须声明和实现另外的接口。这就好像你同样在酒吧点啤酒，为了让服务生理解你的意思，你就必须戴上一顶啤酒杯式的帽子（如图2-4所示），看起来有点多此一举。

图2-4 只想要一杯啤酒，需要这样嘛

通常情况下，这有些让人不好接受。不过，好在这种方式也可以达到目的。

2.2.4 三种注入方式的比较

- **接口注入**。从注入方式的使用上来说，接口注入是现在不甚提倡的一种方式，基本处于“退役状态”。因为它强制被注入对象实现不必要的接口，带有侵入性。而构造方法注入和setter方法注入则不需要如此。
- **构造方法注入**。这种注入方式的优点就是，对象在构造完成之后，即已进入就绪状态，可以马上使用。缺点就是，当依赖对象比较多的时候，构造方法的参数列表会比较长。而通过反射构造对象的时候，对相同类型的参数的处理会比较困难，维护和使用上也比较麻烦。而且在Java中，构造方法无法被继承，无法设置默认值。对于非必须的依赖处理，可能需要引入多个构造方法，而参数数量的变动可能造成维护上的不便。
- **setter方法注入**。因为方法可以命名，所以setter方法注入在描述性上要比构造方法注入好一些。另外，setter方法可以被继承，允许设置默认值，而且有良好的IDE支持。缺点当然就是对象无法在构造完成后马上进入就绪状态。

综上所述，构造方法注入和setter方法注入因为其侵入性较弱，且易于理解和使用，所以是现在使用最多的注入方式；而接口注入因为侵入性较强，近年来已经不流行了。

2.3 IoC 的附加值

从主动获取依赖关系的方式转向IoC方式，不只是一个方向上的改变，简单的转变背后实际上蕴藏着更多的玄机。要说IoC模式能带给我们什么好处，可能各种资料或书籍中已经罗列很多了。比如不会对业务对象构成很强的侵入性，使用IoC后，对象具有更好的可测试性、可重用性和可扩展性，等等。不过，泛泛而谈可能无法真正地让你深刻理解IoC模式带来的诸多好处，所以，还是让我们从具体的示例入手，来一探究竟吧。

对于前面例子中的`FXNewsProvider`来说，在使用IoC重构之前，如果没有其他需求或变动，不光看起来，用起来也是没有问题的。但是，当系统中需要追加逻辑以处理另一家新闻社的新闻来源时，问题就来了。

突然有一天，客户告诉你，我们又搞定一家新闻社，现在可以使用他们的新闻服务了，这家新闻社叫MarketWin24。这个时候，你该如何处理呢？首先，毫无疑问地，应该先根据MarketWin24的服务接口提供一个MarketWin24NewsListener实现，用来接收新闻；其次，因为都是相同的数据访问逻辑，所以原来的DowJonesNewsPersister可以重用，我们先放在一边不管。最后，就主要是业务处理对象FXNewsProvider了。因为我们之前没有用IoC，所以，现在的对象跟DowJonesNewsListener是绑定的，我们无法重用这个类了，不是吗？为了解决问题，我们可能要重新实现一个继承自FXNewsProvider的MarketWin24NewsProvider，或者干脆重新写一个类似的功能。

而使用IoC后，面对同样的需求，我们却完全可以不做任何改动，就直接使用FXNewsProvider。因为不管是DowJones还是MarketWin24，对于我们的系统来说，处理逻辑实际上应该是一样的：根据各个公司的连接接口取得新闻，然后将取得的新闻存入数据库。因此，我们只要根据MarketWin24的新闻服务接口，为MarketWin24的FXNewsProvider提供相应的MarketWin24NewsListener注入就可以了，见代码清单2-5。

代码清单2-5　构建在IoC之上可重用的FXNewsProvider使用演示

```
FXNewsProvider dowJonesNewsProvider = ➥
new FXNewsProvider(new DowJonesNewsListener(),new DowJonesNewsPersister());
...
FXNewsPrivider marketWin24NewsProvider = ➥
new FXNewsProvider(new MarketWin24NewsListener(),new DowJonesNewsPersister());
...
```

看！使用IoC之后，FXNewsProvider可以重用，而不必因为添加新闻来源去重新实现新的FXNewsProvider。实际上，只需要给出特定的IFXNewsListener实现即可。

随着开源项目的成功，TDD（Test Driven Developement，测试驱动开发）已经成为越来越受重视的一种开发方式。因为保证业务对象拥有良好的可测试性，可以为最终交付高质量的软件奠定良好的基础，同时也拉起了产品质量的第一道安全网。所以对于软件开发来说，设计开发可测试性良好的业务对象是至关重要的。而IoC模式可以让我们更容易达到这个目的。比如，使用IoC模式后，为了测试FXNewsProvider，我们可以根据测试的需求，提供一个MockNewsListener给FXNewsProvider。在此之前，我们无法将对DowJonesNewsListener的依赖排除在外，从而导致难以开展单元测试。而现在，单元测试则可以毫无牵绊地进行，代码清单2-6演示了测试取得新闻失败的情形。

代码清单2-6　测试FXNewsProvider类的相关定义

```
测试新闻取得失败的MockNewsListner定义
public class MockNewsListener implements IFXNewsListener
{
  public String[] getAvailableNewsIds() {
    throw new FXNewsRetrieveFailureException();
  }
  public FXNewsBean getNewsByPK(String newsId) {
    // TODO
    return null;
  }
  public void postProcessIfNecessary(String newsId) {
    // TODO
  }
}
相应的FXNewsProvider的单元测试类
```

```
public class FXNewsProviderTest extends TestCase {
  private FXNewsProvider newsProvider;

  @Override
  protected void setUp() throws Exception {
    super.setUp();
    newsProvider = new FXNewsProvider(new MockNewsListener(),new MockNewsPersister());
  }
  @Override
  protected void tearDown() throws Exception {
    super.tearDown();
    newsProvider = null;
  }
  public void testGetAndPersistNewsWithoutResourceAvailable()
  {
    try
    {
      newsProvider.getAndPersistNews();
      fail("Since MockNewsListener has no news support, ➥
      we should fail to get above.");
    }
    catch(FXNewsRetrieveFailureException e)
    {
      //……
    }
  }
}
```

由此可见，相关资料或书籍提到IoC总会赞不绝口，并不是没有原因的。如果你还心存疑虑，那么自己去验证一下吧！说不定你还可以收获更多。毕竟，实践出真知嘛。

如果要用一句话来概括IoC可以带给我们什么，那么我希望是，IoC是一种可以帮助我们解耦各业务对象间依赖关系的对象绑定方式！

2.4 小结

本章主要介绍了IoC或者说依赖注入的概念，讨论了几种基本的依赖注入方式。还与大家一起探索并验证了IoC所带给我们的部分“附加值”。所以，现在大家应该对IoC或者说依赖注入有了最基本认识。下一章，我们将一起去更深入地了解IoC场景中的重要角色，即IoC Service Provider。

第 3 章

掌管大局的IoC Service Provider

本章内容

- IoC Service Provider的职责
- 运筹帷幄的秘密——IoC Service Provider如何管理对象间的依赖关系

虽然业务对象可以通过IoC方式声明相应的依赖，但是最终仍然需要通过某种角色或者服务将这些相互依赖的对象绑定到一起，而IoC Service Provider就对应IoC场景中的这一角色。

IoC Service Provider在这里是一个抽象出来的概念，它可以指代任何将IoC场景中的业务对象绑定到一起的实现方式。它可以是一段代码，也可以是一组相关的类，甚至可以是比较通用的IoC框架或者IoC容器实现。比如，可以通过以下代码（见代码清单3-1）绑定与新闻相关的对象。

代码清单3-1　FXNewsProvider相关依赖绑定代码

```
IFXNewsListener newsListener = new DowJonesNewsListener();
IFXNewsPersister newsPersister = new DowJonesNewsPersister();
FXNewsProvider newsProvider = new FXNewsProvider(newsListener,newsPersister);
newsProvider.getAndPersistNews();
```

这段代码就可以认为是这个场景中的IoC Service Provider，只不过比较简单，而且目的也过于单一罢了。要将系统中几十、几百甚至数以千计的业务对象绑定到一起，采用这种方式显然是不切实际的。通用性暂且不提，单单是写这些绑定代码也会是一种很糟糕的体验。不过，好在现在许多开源产品通过各种方式为我们做了这部分工作。所以，目前来看，我们只需要使用这些产品提供的服务就可以了。Spring的IoC容器就是一个提供依赖注入服务的IoC Service Provider。

3.1　IoC Service Provider 的职责

IoC Service Provider的职责相对来说比较简单，主要有两个：业务对象的构建管理和业务对象间的依赖绑定。

- **业务对象的构建管理**。在IoC场景中，业务对象无需关心所依赖的对象如何构建如何取得，但这部分工作始终需要有人来做。所以，IoC Service Provider需要将对象的构建逻辑从客户端对象[①]那里剥离出来，以免这部分逻辑污染业务对象的实现。
- **业务对象间的依赖绑定**。对于IoC Service Provider来说，这个职责是最艰巨也是最重要的，这是它的最终使命之所在。如果不能完成这个职责，那么，无论业务对象如何的“呼喊”，也不会得到依赖对象的任何响应（最常见的倒是会收到一个NullPointerException）。IoC Service Provider通过结合之前构建和管理的所有业务对象，以及各个业务对象间可以识别的依赖关系，

① 这里指代使用某个对象或者某种服务的对象。如果对象A需要引用对象B，那么A就是B的客户端对象，而不管A处于Service层还是数据访问层。

将这些对象所依赖的对象注入绑定，从而保证每个业务对象在使用的时候，可以处于就绪状态。

3.2　运筹帷幄的秘密——IoC Service Provider 如何管理对象间的依赖关系

前面我们说过，被注入对象可以通过多种方式通知IoC Service Provider为其注入相应依赖。但问题在于，收到通知的IoC Service Provider是否就一定能够完全领会被注入对象的意图，并及时有效地为其提供想要的依赖呢？有些时候，事情可能并非像我们所想象的那样理所当然。

还是拿酒吧的例子说事儿，不管是常客还是初次光顾，你都可以点自己需要的饮料，以任何方式通知服务生都可以。要是侍者经验老道，你需要的任何饮品他都知道如何为你调制并提供给你。可是，如果服务生刚入行又会如何呢？当他连啤酒、鸡尾酒都分不清的时候，你能指望他及时地将你需要的饮品端上来吗？

服务生最终必须知道顾客点的饮品与库存饮品的对应关系，才能为顾客端上适当的饮品。对于为被注入对象提供依赖注入的IoC Service Provider来说，它也同样需要知道自己所管理和掌握的被注入对象和依赖对象之间的对应关系。

IoC Service Provider不是人类，也就不能像酒吧服务生那样通过大脑来记忆和存储所有的相关信息。所以，它需要寻求其他方式来记录诸多对象之间的对应关系。比如：

- 它可以通过最基本的文本文件来记录被注入对象和其依赖对象之间的对应关系；
- 它也可以通过描述性较强的XML文件格式来记录对应信息；
- 它还可以通过编写代码的方式来注册这些对应信息；
- 甚至，如果愿意，它也可以通过语音方式来记录对象间的依赖注入关系（“嗨，它要一个这种类型的对象，拿这个给它”）。

那么，实际情况下，各种具体的IoC Service Provider实现又是通过哪些方式来记录“服务信息”的呢？

我们可以归纳一下，当前流行的IoC Service Provider产品使用的注册对象管理信息的方式主要有以下几种。

3.2.1　直接编码方式

当前大部分的IoC容器都应该支持直接编码方式，比如PicoContainer[①]、Spring、Avalon等。在容器启动之前，我们就可以通过程序编码的方式将被注入对象和依赖对象注册到容器中，并明确它们相互之间的依赖注入关系。代码清单3-2中的伪代码演示了这样一个过程。

代码清单3-2　直接编码方式管理对象间的依赖注入关系

```
IoContainer container = ...;
container.register(FXNewsProvider.class,new FXNewsProvider());
container.register(IFXNewsListener.class,new DowJonesNewsListener());
...
FXNewsProvider newsProvider = (FXNewsProvider)container.get(FXNewsProvider.class);
newProvider.getAndPersistNews();
```

通过为相应的类指定对应的具体实例，可以告知IoC容器，当我们要这种类型的对象实例时，请将容器中注册的、对应的那个具体实例返回给我们。

① www.picocontainer.org。

如果是接口注入，可能伪代码看起来要多一些。不过，道理上是一样的，只不过除了注册相应对象，还要将“注入标志接口”与相应的依赖对象绑定一下，才能让容器最终知道是一个什么样的对应关系，如代码清单3-3所演示的那样。

代码清单3-3 直接编码形式管理基于接口注入的依赖注入关系

```
IoContainer container = ...;
container.register(FXNewsProvider.class,new FXNewsProvider());
container.register(IFXNewsListener.class,new DowJonesNewsListener());
...
container.bind(IFXNewsListenerCallable.class, container.get(IFXNewsListener.class));
...
FXNewsProvider newsProvider = (FXNewsProvider)container.get(FXNewsProvider.class);
newProvider.getAndPersistNews();
```

通过bind方法将“被注入对象”（由IFXNewsListenerCallable接口添加标志）所依赖的对象，绑定为容器中注册过的IFXNewsListener类型的对象实例。容器在返回FXNewsProvider对象实例之前，会根据这个绑定信息，将IFXNewsListener注册到容器中的对象实例注入到“被注入对象”——FXNewsProvider中，并最终返回已经组装完毕的FXNewsProvider对象。

所以，通过程序编码让最终的IoC Service Provider（也就是各个IoC框架或者容器实现）得以知晓服务的“奥义”，应该是管理依赖绑定关系的最基本方式。

3.2.2 配置文件方式

这是一种较为普遍的依赖注入关系管理方式。像普通文本文件、properties文件、XML文件等，都可以成为管理依赖注入关系的载体。不过，最为常见的，还是通过XML文件来管理对象注册和对象间依赖关系，比如Spring IoC容器和在PicoContainer基础上扩展的NanoContainer，都是采用XML文件来管理和保存依赖注入信息的。对于我们例子中的FXNewsProvider来说，也可以通过Spring配置文件的方式（见代码清单3-4）来配置和管理各个对象间的依赖关系。

代码清单3-4 通过Spring的配置方式来管理FXNewsProvider的依赖注入关系

```
<bean id="newsProvider" class="..FXNewsProvider">
  <property name="newsListener">
    <ref bean="djNewsListener"/>
  </property>
  <property name="newPersistener">
    <ref bean="djNewsPersister"/>
  </property>
</bean>

<bean id="djNewsListener"
  class="..impl.DowJonesNewsListener">
</bean>
<bean id="djNewsPersister"
  class="..impl.DowJonesNewsPersister">
</bean>
```

最后，我们就可以像代码清单3-5所示的那样，通过“newsProvider”这个名字，从容器中取得已经组装好的FXNewsProvider并直接使用。

代码清单3-5 从读取配置文件完成对象组装的容器中获取FXNewsProvider并使用

```
...
container.readConfigurationFiles(...);
FXNewsProvider newsProvider = (FXNewsProvider)container.getBean("newsProvider");
newsProvider.getAndPersistNews();
```

3.2.3　元数据方式

这种方式的代表实现是Google Guice，这是Bob Lee在Java 5的注解和Generic的基础上开发的一套IoC框架。我们可以直接在类中使用元数据信息来标注各个对象之间的依赖关系，然后由Guice框架根据这些注解所提供的信息将这些对象组装后，交给客户端对象使用。代码清单3-6演示了使用Guice的相应注解标注后的FXNewsProvider定义。

代码清单3-6　使用Guice的注解标注依赖关系后的FXNewsProvider定义

```
public class FXNewsProvider
{
  private IFXNewsListener  newsListener;
  private IFXNewsPersister newPersistener;
  @Inject
  public FXNewsProvider(IFXNewsListener listener,IFXNewsPersister persister)
  {
    this.newsListener   = listener;
    this.newPersistener = persister;
  }
  ...
}
```

通过@Inject，我们指明需要IoC Service Provider通过构造方法注入方式，为FXNewsProvider注入其所依赖的对象。至于余下的依赖相关信息，在Guice中是由相应的Module来提供的，代码清单3-7给出了FXNewsProvider所使用的Module实现。

代码清单3-7　FXNewsProvider所使用的Module实现

```
public class NewsBindingModule extends AbstractModule
{
  @Override
  protected void configure() {
    bind(IFXNewsListener.class) ➥
    .to(DowJonesNewsListener.class).in(Scopes.SINGLETON);
    bind(IFXNewsPersister.class) ➥
    .to(DowJonesNewsPersister.class).in(Scopes.SINGLETON);
  }
}
```

通过Module指定进一步的依赖注入相关信息之后，我们就可以直接从Guice那里取得最终已经注入完毕，并直接可用的对象了（见代码清单3-8）。

代码清单3-8　从Guice获取并使用最终绑定完成的FXNewsProvider

```
Injector injector = Guice.createInjector(new NewsBindingModule());
FXNewsProvider newsProvider = injector.getInstance(FXNewsProvider.class);
newsProvider.getAndPersistNews();
```

当然，注解最终也要通过代码处理来确定最终的注入关系，从这点儿来说，注解方式可以算作编码方式的一种特殊情况。

3.3　小结

本章就IoC场景中的主要角色IoC Service Provider给出了言简意赅的介绍。讨论了IoC Service Provider的基本职责，以及它常用的几种依赖关系管理方式。

应该说，IoC Service Provider只是为了简化概念而提出的一个一般性的概念。下一章，我们将由一般到特殊，一起深入了解一个特定的IoC Service Provider实现产品，即Spring提供的IoC容器。

第4章 Spring的IoC容器之BeanFactory

本章内容

- 拥有BeanFactory之后的生活
- BeanFactory的对象注册与依赖绑定方式
- BeanFactory的XML之旅
- 容器背后的秘密

我们前面说过，Spring的IoC容器是一个IoC Service Provider，但是，这只是它被冠以IoC之名的部分原因，我们不能忽略的是“容器”。Spring的IoC容器是一个提供IoC支持的轻量级容器，除了基本的IoC支持，它作为轻量级容器还提供了IoC之外的支持。如在Spring的IoC容器之上，Spring还提供了相应的AOP框架支持、企业级服务集成等服务。Spring的IoC容器和IoC Service Provider所提供的服务之间存在一定的交集，二者的关系如图4-1所示。

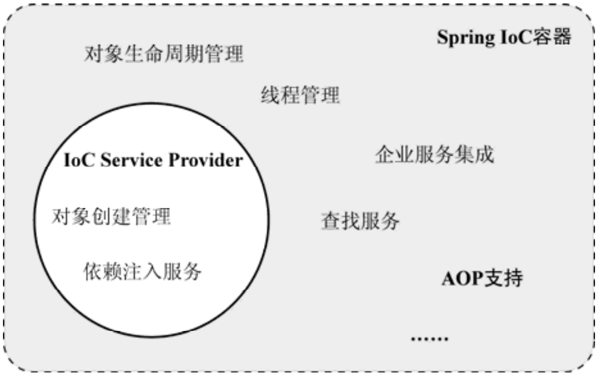

图4-1 Spring的IoC容器和IoC Service Provider之间的关系

> **注意** 本章将主要关注Spring的IoC容器提供的IoC相关支持以及衍生的部分高级特性。而IoC容器提供的其他服务将在后继章节中陆续阐述。

Spring提供了两种容器类型：BeanFactory和ApplicationContext。

- BeanFactory。基础类型IoC容器，提供完整的IoC服务支持。如果没有特殊指定，默认采用延迟初始化策略（lazy-load）。只有当客户端对象需要访问容器中的某个受管对象的时候，才对该受管对象进行初始化以及依赖注入操作。所以，相对来说，容器启动初期速度较快，所需要的资源有限。对于资源有限，并且功能要求不是很严格的场景，BeanFactory是比较合适的IoC容器选择。
- ApplicationContext。ApplicationContext在BeanFactory的基础上构建，是相对比较高级的容器实现，除了拥有BeanFactory的所有支持，ApplicationContext还提供了其他高级

特性，比如事件发布、国际化信息支持等，这些会在后面详述。ApplicationContext所管理的对象，在该类型容器启动之后，默认全部初始化并绑定完成。所以，相对于BeanFactory来说，ApplicationContext要求更多的系统资源，同时，因为在启动时就完成所有初始化，容器启动时间较之BeanFactory也会长一些。在那些系统资源充足，并且要求更多功能的场景中，ApplicationContext类型的容器是比较合适的选择。

通过图4-2，我们可以对BeanFactory和ApplicationContext之间的关系有一个更清晰的认识。

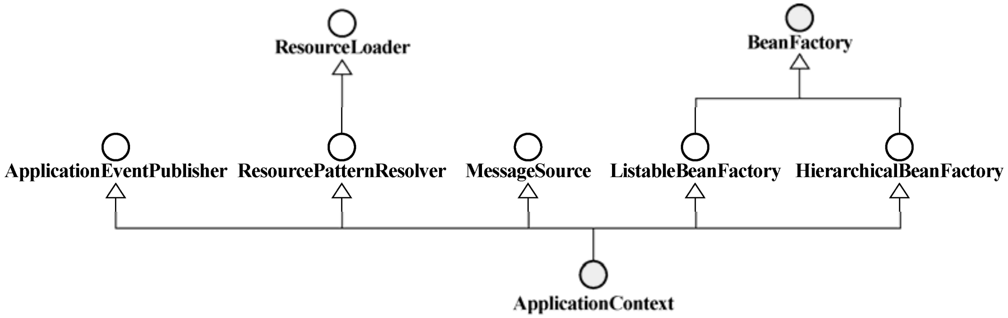

图4-2　BeanFactory和ApplicationContext继承关系

> **注意**　ApplicationContext间接继承自BeanFactory，所以说它是构建于BeanFactory之上的IoC容器。此外，你应该注意到了，ApplicationContext还继承了其他三个接口，它们之间的关系，我们将在第5章中详细说明。
>
> 另外，在没有特殊指明的情况下，以BeanFactory为中心所讲述的内容同样适用于ApplicationContext，这一点需要明确一下，二者有差别的地方会在合适的位置给出解释。

BeanFactory，顾名思义，就是生产Bean的工厂。当然，严格来说，这个“生产过程”可能不像说起来那么简单。既然Spring框架提倡使用POJO，那么把每个业务对象看作一个JavaBean对象，或许更容易理解为什么Spring的IoC基本容器会起这么一个名字。作为Spring提供的基本的IoC容器，BeanFactory可以完成作为IoC Service Provider的所有职责，包括业务对象的注册和对象间依赖关系的绑定。

BeanFactory就像一个汽车生产厂。你从其他汽车零件厂商或者自己的零件生产部门取得汽车零件送入这个汽车生产厂，最后，只需要从生产线的终点取得成品汽车就可以了。相似地，将应用所需的所有业务对象交给BeanFactory之后，剩下要做的，就是直接从BeanFactory取得最终组装完成并且可用的对象。至于这个最终业务对象如何组装，你不需要关心，BeanFactory会帮你搞定。

所以，对于客户端来说，与BeanFactory打交道其实很简单。最基本地，BeanFactory肯定会公开一个取得组装完成的对象的方法接口，就像代码清单4-1中真正的BeanFactory的定义所展示的那样。

代码清单4-1　BeanFactory的定义

```
public interface BeanFactory {
   String FACTORY_BEAN_PREFIX = "&";
   Object getBean(String name) throws BeansException;
```

```
    Object getBean(String name, Class requiredType) throws BeansException;
    /**
     * @since 2.5
     */
    Object getBean(String name, Object[] args) throws BeansException;
    boolean containsBean(String name);
    boolean isSingleton(String name) throws NoSuchBeanDefinitionException;
    /**
     * @since 2.0.3
     */
    boolean isPrototype(String name) throws NoSuchBeanDefinitionException;
    /**
     * @since 2.0.1
     */
    boolean isTypeMatch(String name, Class targetType) throws NoSuchBeanDefinitionException;
    Class getType(String name) throws NoSuchBeanDefinitionException;
    String[] getAliases(String name);
}
```

上面代码中的方法基本上都是查询相关的方法，例如，取得某个对象的方法（getBean）、查询某个对象是否存在于容器中的方法（containsBean），或者取得某个bean的状态或者类型的方法等。因为通常情况下，对于独立的应用程序，只有主入口类才会跟容器的API直接耦合。

4.1　拥有BeanFactory之后的生活

确切地说，拥有BeanFactory之后的生活没有太大的变化。当然，我的意思是看起来没有太大的变化。到底引入BeanFactory后的生活是什么样子，让我们一起来体验一下吧！

依然“拉拉扯扯的事情”。对于应用程序的开发来说，不管是否引入BeanFactory之类的轻量级容器，应用的设计和开发流程实际上没有太大改变。换句话说，针对系统和业务逻辑，该如何设计和实现当前系统不受是否引入轻量级容器的影响。对于我们的FX新闻系统，我们还是会针对系统需求，分别设计相应的接口和实现类。前后唯一的不同，就是对象之间依赖关系的解决方式改变了。这就是所谓的“拉拉扯扯的事情”。之前我们的系统业务对象需要自己去“拉”（Pull）所依赖的业务对象，有了BeanFactory之类的IoC容器之后，需要依赖什么让BeanFactory为我们推过来（Push）就行了。所以，简单点儿说，拥有BeanFactory之后，要使用IoC模式进行系统业务对象的开发。（实际上，即使不使用BeanFactory之类的轻量级容器支持开发，开发中也应该尽量使用IoC模式。）代码清单4-2演示了FX新闻系统初期的设计和实现框架代码。

代码清单4-2　FX新闻应用设计和实现框架代码

```
1-设计FXNewsProvider类用于普遍的新闻处理
public class FXNewsProvider
{
  ...
}

2-设计IFXNewsListener接口抽象各个新闻社不同的新闻获取方式，并给出相应实现类
public interface IFXNewsListener
{
  ...
}
以及
public class DowJonesNewsListener implements IFXNewsListener
```

```
{
   ...
}

3-设计IFXNewsPersister接口抽象不同数据访问方式，并实现相应的实现类
public interface IFXNewsPersister
{
   ...
}
以及
public class DowJonesNewsPersister implements IFXNewsPersister
{
   ...
}
```

BeanFactory会说，这些让我来干吧。既然使用IoC模式开发的业务对象现在不用自己操心如何解决相互之间的依赖关系，那么肯定得找人来做这个工作。毕竟，工作最终是要有人来做的，大家都不动手，那工作就不能进行了。当BeanFactory说这些事情让它来做的时候，可能没有告诉你它会怎么来做这个事情。不过没关系，稍后我会详细告诉你它是如何做的。通常情况下，它会通过常用的XML文件来注册并管理各个业务对象之间的依赖关系，就像代码清单4-3所演示的那样。

代码清单4-3 使用BeanFactory的XML配置方式实现业务对象间的依赖管理

```
<beans>
  <bean id="djNewsProvider" class="..FXNewsProvider">
    <constructor-arg index="0">
      <ref bean="djNewsListener"/>
    </constructor-arg>
    <constructor-arg index="1">
      <ref bean="djNewsPersister"/>
    </constructor-arg>
  </bean>
  ...
</beans>
```

拉响启航的汽笛。在BeanFactory出现之前，我们通常会直接在应用程序的入口类的main方法中，自己实例化相应的对象并调用之，如以下代码所示：

```
FXNewsProvider newsProvider = new FXNewsProvider();
newsProvider.getAndPersistNews();
```

不过，现在既然有了BeanFactory，我们通常只需将“生产线图纸”交给BeanFactory，让BeanFactory为我们生产一个FXNewsProvider，如以下代码所示：

```
BeanFactory container = ➥
new XmlBeanFactory(new ClassPathResource("配置文件路径"));
FXNewsProvider newsProvider = (FXNewsProvider)container.getBean("djNewsProvider");
newsProvider.getAndPersistNews();
```

或者如以下代码所示：

```
ApplicationContext container = ➥
new ClassPathXmlApplicationContext("配置文件路径");
FXNewsProvider newsProvider = (FXNewsProvider)container.getBean("djNewsProvider");
newsProvider.getAndPersistNews();
```

亦或如以下代码所示：

```
ApplicationContext container = ➥
```

```
new FileSystemXmlApplicationContext("配置文件路径");
FXNewsProvider newsProvider = (FXNewsProvider)container.getBean("djNewsProvider");
newsProvider.getAndPersistNews();
```

这就是拥有BeanFactory后的生活。当然，这只是使用BeanFactory后开发流程的一个概览而已，具体细节请容我慢慢道来。

4.2　BeanFactory的对象注册与依赖绑定方式①

BeanFactory作为一个IoC Service Provider，为了能够明确管理各个业务对象以及业务对象之间的依赖绑定关系，同样需要某种途径来记录和管理这些信息。上一章在介绍IoC Service Provider时，我们提到通常会有三种方式来管理这些信息。而BeanFactory几乎支持所有这些方式，很令人兴奋，不是吗？

4.2.1　直接编码方式

其实，把编码方式单独提出来称作一种方式并不十分恰当。因为不管什么方式，最终都需要编码才能“落实”所有信息并付诸使用。不过，通过这些代码，起码可以让我们更加清楚BeanFactory在底层是如何运作的。

下面来看一下我们的FX新闻系统相关类是如何注册并绑定的（见代码清单4-4）。

代码清单4-4　通过编码方式使用BeanFactory实现FX新闻相关类的注册及绑定

```
public static void main(String[] args)
{
  DefaultListableBeanFactory beanRegistry = new DefaultListableBeanFactory();
  BeanFactory container = (BeanFactory)bindViaCode(beanRegistry);
  FXNewsProvider newsProvider = ➥
  (FXNewsProvider)container.getBean("djNewsProvider");
  newsProvider.getAndPersistNews();
}

public static BeanFactory bindViaCode(BeanDefinitionRegistry registry)
{
    AbstractBeanDefinition newsProvider = ➥
    new RootBeanDefinition(FXNewsProvider.class,true);
    AbstractBeanDefinition newsListener = ➥
    new RootBeanDefinition(DowJonesNewsListener.class,true);
    AbstractBeanDefinition newsPersister = ➥
    new RootBeanDefinition(DowJonesNewsPersister.class,true);
    // 将bean定义注册到容器中
    registry.registerBeanDefinition("djNewsProvider", newsProvider);
    registry.registerBeanDefinition("djListener", newsListener);
    registry.registerBeanDefinition("djPersister", newsPersister);
// 指定依赖关系
    // 1. 可以通过构造方法注入方式
    ConstructorArgumentValues argValues = new ConstructorArgumentValues();
    argValues.addIndexedArgumentValue(0, newsListener);
    argValues.addIndexedArgumentValue(1, newsPersister);
    newsProvider.setConstructorArgumentValues(argValues);
    // 2. 或者通过setter方法注入方式
```

① 在Spring的术语中，把BeanFactory需要使用的对象注册和依赖绑定信息称为Configuration Metadata。我们这里所展示的，实际上就是组织这些Configuration Metadata的各种方式。因此这个标题才这么长。

```
    MutablePropertyValues propertyValues = new MutablePropertyValues();
    propertyValues.addPropertyValue(new ropertyValue("newsListener",newsListener));
    propertyValues.addPropertyValue(new PropertyValue("newPersistener",newsPersister));
    newsProvider.setPropertyValues(propertyValues);
    // 绑定完成
    return (BeanFactory)registry;
}
```

BeanFactory只是一个接口，我们最终需要一个该接口的实现来进行实际的Bean的管理，DefaultListableBeanFactory就是这么一个比较通用的BeanFactory实现类。DefaultListableBeanFactory除了间接地实现了BeanFactory接口，还实现了BeanDefinitionRegistry接口，该接口才是在BeanFactory的实现中担当Bean注册管理的角色。基本上，BeanFactory接口只定义如何访问容器内管理的Bean的方法，各个BeanFactory的具体实现类负责具体Bean的注册以及管理工作。BeanDefinitionRegistry接口定义抽象了Bean的注册逻辑。通常情况下，具体的BeanFactory实现类会实现这个接口来管理Bean的注册。它们之间的关系如图4-3所示。

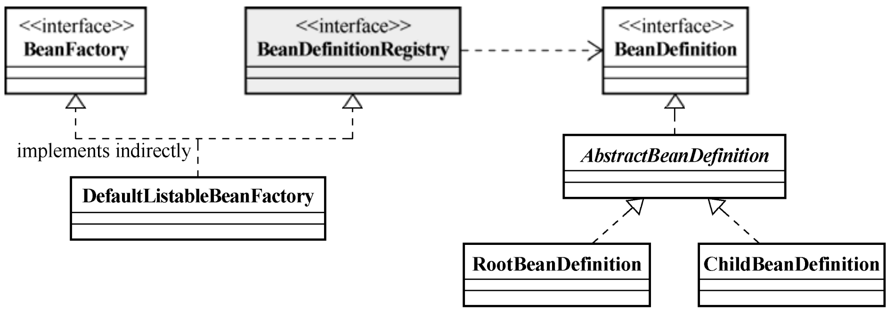

图4-3 BeanFactory、BeanDefinitionRegistry以及DefaultListableBeanFactory的关系

打个比方说，BeanDefinitionRegistry就像图书馆的书架，所有的书是放在书架上的。虽然你还书或者借书都是跟图书馆（也就是BeanFactory，或许BookFactory可能更好些）打交道，但书架才是图书馆存放各类图书的地方。所以，书架相对于图书馆来说，就是它的“BookDefinitionRegistry”。

每一个受管的对象，在容器中都会有一个BeanDefinition的实例（instance）与之相对应，该BeanDefinition的实例负责保存对象的所有必要信息，包括其对应的对象的class类型、是否是抽象类、构造方法参数以及其他属性等。当客户端向BeanFactory请求相应对象的时候，BeanFactory会通过这些信息为客户端返回一个完备可用的对象实例。RootBeanDefinition和ChildBeanDefinition是BeanDefinition的两个主要实现类。

现在，我们再来看这段绑定代码，应该就有“柳暗花明”的感觉了。

- 在main方法中，首先构造一个DefaultListableBeanFactory作为BeanDefinitionRegistry，然后将其交给bindViaCode方法进行具体的对象注册和相关依赖管理，然后通过bindViaCode返回的BeanFactory取得需要的对象，最后执行相应逻辑。在我们的实例里，当然就是取得FXNewsProvider进行新闻的处理。
- 在bindViaCode方法中，首先针对相应的业务对象构造与其相对应的BeanDefinition，使用了RootBeanDefinition作为BeanDefinition的实现类。构造完成后，将这些BeanDefinition注册到通过方法参数传进来的BeanDefinitionRegistry中。之后，因为我们的FXNewsProvider是采用的构造方法注入，所以，需要通过ConstructorArgument-

Values为其注入相关依赖。在这里为了同时说明setter方法注入，也同时展示了在Spring中如何使用代码实现setter方法注入。如果要运行这段代码，需要把setter方法注入部分的4行代码注释掉。最后，以BeanFactory的形式返回已经注册并绑定了所有相关业务对象的BeanDefinitionRegistry实例。

小心 最后一行的强制类型转换是有特定场景的。因为传入的DefaultListableBeanFactory同时实现了BeanFactory和BeanDefinitionRegistry接口，所以，这样做强制类型转换不会出现问题。但需要注意的是，单纯的BeanDefinitionRegistry是无法强制转换到BeanFactory类型的！

4.2.2　外部配置文件方式

Spring的IoC容器支持两种配置文件格式：Properties文件格式和XML文件格式。当然，如果你愿意也可以引入自己的文件格式，前提是真的需要。

采用外部配置文件时，Spring的IoC容器有一个统一的处理方式。通常情况下，需要根据不同的外部配置文件格式，给出相应的BeanDefinitionReader实现类，由BeanDefinitionReader的相应实现类负责将相应的配置文件内容读取并映射到BeanDefinition，然后将映射后的BeanDefinition注册到一个BeanDefinitionRegistry，之后，BeanDefinitionRegistry即完成Bean的注册和加载。当然，大部分工作，包括解析文件格式、装配BeanDefinition之类的工作，都是由BeanDefinitionReader的相应实现类来做的，BeanDefinitionRegistry只不过负责保管而已。整个过程类似于如下代码：

```
BeanDefinitionRegistry beanRegistry = <某个BeanDefinitionRegistry实现类，通常为➥
DefaultListableBeanFactory>;
BeanDefinitionReader  beanDefinitionReader = new BeanDefinitionReaderImpl(beanRegistry);
beanDefinitionReader.loadBeanDefinitions("配置文件路径");
// 现在我们就取得了一个可用的BeanDefinitionRegistry实例
```

1. Properties配置格式的加载

Spring提供了org.springframework.beans.factory.support.PropertiesBeanDefinitionReader类用于Properties格式配置文件的加载，所以，我们不用自己去实现BeanDefinitionReader，只要根据该类的读取规则，提供相应的配置文件即可。

对于FXNews系统的业务对象，我们采用如下文件内容（见代码清单4-5）进行配置加载。

代码清单4-5　Properties格式表达的依赖注入配置内容

```
djNewsProvider.(class)=..FXNewsProvider
# ----------通过构造方法注入的时候-------------
djNewsProvider.$0(ref)=djListener
djNewsProvider.$1(ref)=djPersister
# ----------通过setter方法注入的时候---------
# djNewsProvider.newsListener(ref)=djListener
# djNewsProvider.newPersistener(ref)=djPersister

djListener.(class)=..impl.DowJonesNewsListener

djPersister.(class)=..impl.DowJonesNewsPersister
```

这些内容是特定于Spring的PropertiesBeanDefinitionReader的，要了解更多内容，请参照

Spring的API参考文档。我们可以很容易地看明白代码清单4-5中的配置内容所要表达的意思。

- djNewsProvider作为beanName，后面通过.(class)表明对应的实现类是什么，实际上使用djNewsProvider.class=...的形式也是可以的，但Spring 1.2.6之后不再提倡使用，而提倡使用.(class)的形式。其他两个类的注册，djListener和djPersister，也是相同的道理。
- 通过在表示beanName的名称后添加.$[number]后缀的形式，来表示当前beanName对应的对象需要通过构造方法注入的方式注入相应依赖对象。在这里，我们分别将构造方法的第一个参数和第二个参数对应到djListener和djPersister。需要注意的一点，就是$0和$1后面的(ref)，(ref)用来表示所依赖的是引用对象，而不是普通的类型。如果不加(ref)，PropertiesBeanDefinitionReader会将djListener和djPersister作为简单的String类型进行注入，异常自然不可避免啦。
- FXNewsProvider采用的是构造方法注入，而为了演示setter方法注入在Properties配置文件中又是一个什么样子，以便于你更全面地了解基于Properties文件的配置方式，我们在下面增加了setter方法注入的例子，不过进行了注释。实际上，与构造方法注入最大的区别就是，它不使用数字顺序来指定注入的位置，而使用相应的属性名称来指定注入。newsListener和newPersistener恰好就是我们的FXNewsProvider类中所声明的属性名称。这印证了之前在比较构造方法注入和setter方法注入方式不同时提到的差异，即构造方法注入无法通过参数名称来标识注入的确切位置，而setter方法注入则可以通过属性名称来明确标识注入。与在Properties中表达构造方法注入一样，同样需要注意，如果属性名称所依赖的是引用对象，那么一定不要忘了(ref)。

当这些对象之间的注册和依赖注入信息都表达清楚之后，就可以将其加载到BeanFactory而付诸使用了。而这个加载过程实际上也就像我们之前总体上所阐述的那样，代码清单4-6中的内容再次演示了类似的加载过程。

代码清单4-6　加载Properties配置的BeanFactory的使用演示

```
public static void main(String[] args)
{
  DefaultListableBeanFactory beanRegistry = new DefaultListableBeanFactory();
  BeanFactory container = (BeanFactory)bindViaPropertiesFile(beanRegistry);
  FXNewsProvider newsProvider =
(FXNewsProvider)container.getBean("djNewsProvider");
  newsProvider.getAndPersistNews();
}

public static BeanFactory bindViaPropertiesFile(BeanDefinitionRegistry registry)
{
  PropertiesBeanDefinitionReader reader =
new PropertiesBeanDefinitionReader(registry);
  reader.loadBeanDefinitions("classpath:../../binding-config.properties");
  return (BeanFactory)registry;
}
```

基于Properties的加载方式就是这么简单，所有的信息配置到Properties文件即可，不用再通过冗长的代码来完成对象的注册和依赖绑定。这些工作就交给相应的BeanDefinitionReader来做吧！哦，我的意思是，让给PropertiesBeanDefinitionReader来做。

注意　Spring提供的PropertiesBeanDefinitionReader是按照Spring自己的文件配置规则进行加载的，而同样的道理，你也可以按照自己的规则[①]来提供相应的Properties配置文件。只不过，现在需要实现你自己的“PropertiesBeanDefinitionReader”来读取并解析。这当然有“重新发明轮子”之嫌，不过，如果你只是想试验一下，也可以尝试哦。无非就是按照自己的规则把各个业务对象信息读取后，将编码方式的代码改造一下放到你自己的“PropertiesBean-DefinitionReader”而已。

2. XML配置格式的加载

XML配置格式是Spring支持最完整，功能最强大的表达方式。当然，一方面这得益于XML良好的语意表达能力；另一方面，就是Spring框架从开始就自始至终保持XML配置加载的统一性。同Properties配置加载类似，现在只不过是转而使用XML而已。Spring 2.*x*之前，XML配置文件采用DTD（Document Type Definition）实现文档的格式约束。2.*x*之后，引入了基于XSD（XML Schema Definition）的约束方式。不过，原来的基于DTD的方式依然有效，因为从DTD转向XSD只是“形式”上的转变，所以，后面的大部分讲解还会沿用DTD的方式，只有必要时才会给出特殊说明。

如果FX新闻系统对象按照XML配置方式进行加载的话，配置文件内容如代码清单4-7所示。

代码清单4-7　FX新闻系统相关类对应XML格式的配置内容

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" ➥
"http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
  <bean id="djNewsProvider" class="..FXNewsProvider">
    <constructor-arg index="0">
      <ref bean="djNewsListener"/>
    </constructor-arg>
    <constructor-arg index="1">
      <ref bean="djNewsPersister"/>
    </constructor-arg>
  </bean>

  <bean id="djNewsListener" class="..impl.DowJonesNewsListener">
  </bean>
  <bean id="djNewsPersister" class="..impl.DowJonesNewsPersister">
  </bean>
</beans>
```

我想这段内容不需要特殊说明吧，应该比Properties文件的内容要更容易理解。如果想知道这些内容背后的更多玄机，往后看吧！

有了XML配置文件，我们需要将其内容加载到相应的BeanFactory实现中，以供使用，如代码清单4-8所示。

代码清单4-8　加载XML配置文件的BeanFactory的使用演示

```
public static void main(String[] args)
{
```

① 也就是，如果我们不满足于Spring的默认规则，可以实现自己喜欢的规则表达方式。

```
    DefaultListableBeanFactory beanRegistry = new DefaultListableBeanFactory();
    BeanFactory container = (BeanFactory)bindViaXMLFile(beanRegistry);
    FXNewsProvider newsProvider = ➥
    (FXNewsProvider)container.getBean("djNewsProvider");
    newsProvider.getAndPersistNews();
}

public static BeanFactory bindViaXMLFile(BeanDefinitionRegistry registry)
{
    XmlBeanDefinitionReader reader = new XmlBeanDefinitionReader(registry);
    reader.loadBeanDefinitions("classpath:../news-config.xml");
    return (BeanFactory)registry;
    // 或者直接
    //return new XmlBeanFactory(new ClassPathResource("../news-config.xml"));
}
```

与为Properties配置文件格式提供`PropertiesBeanDefinitionReader`相对应，Spring同样为XML格式的配置文件提供了现成的`BeanDefinitionReader`实现，即`XmlBeanDefinitionReader`。`XmlBeanDefinitionReader`负责读取Spring指定格式的XML配置文件并解析，之后将解析后的文件内容映射到相应的`BeanDefinition`，并加载到相应的`BeanDefinitionRegistry`中（在这里是`DefaultListableBeanFactory`）。这时，整个`BeanFactory`就可以放给客户端使用了。

除了提供`XmlBeanDefinitionReader`用于XML格式配置文件的加载，Spring还在`DefaultListableBeanFactory`的基础上构建了简化XML格式配置加载的`XmlBeanFactory`实现。从以上代码最后注释掉的一行，你可以看到使用了`XmlBeanFactory`之后，完成XML的加载和`BeanFactory`的初始化是多么简单。

注意 当然，如果你愿意，就像Properties方式可以扩展一样，XML方式的加载同样可以扩展。虽然`XmlBeanFactory`基本上已经十分完备了，但如果出于某种目的，`XmlBeanFactory`或者默认的`XmlBeanDefinitionReader`所使用的XML格式无法满足需要的话，你同样可以通过扩展`XmlBeanDefinitionReader`或者直接实现自己的`BeanDefinitionReader`来达到自定义XML配置文件加载的目的。Spring的可扩展性为你服务！

4.2.3 注解方式

可能你没有注意到，我在提到`BeanFactory`所支持的对象注册与依赖绑定方式的时候，说的是`BeanFactory`“几乎”支持IoC Service Provider可能使用的所有方式。之所以这么说，有两个原因。

- 在Spring 2.5发布之前，Spring框架并没有正式支持基于注解方式的依赖注入；
- Spring 2.5发布的基于注解的依赖注入方式，如果不使用classpath-scanning功能的话，仍然部分依赖于“基于XML配置文件”的依赖注入方式。

另外，注解是Java 5之后才引入的，所以，以下内容只适用于应用程序使用了Spring 2.5以及Java 5或者更高版本的情况之下。

如果要通过注解标注的方式为`FXNewsProvider`注入所需要的依赖，现在可以使用`@Autowired`以及`@Component`对相关类进行标记。代码清单4-9演示了FXNews相关类使用指定注解标注后的情况。

代码清单4-9　使用指定注解标注后的FXNews相关类

```
@Component
public class FXNewsProvider
```

```
{
  @Autowired
  private IFXNewsListener  newsListener;
  @Autowired
  private IFXNewsPersister newPersistener;

  public FXNewsProvider(IFXNewsListener newsListner,IFXNewsPersister newsPersister)
  {
    this.newsListener   = newsListner;
    this.newPersistener = newsPersister;
  }
  ...
}

@Component
public class DowJonesNewsListener implements IFXNewsListener
{
  ...
}

@Component
public class DowJonesNewsPersister implements IFXNewsPersister
{
  ...
}
```

@Autowired是这里的主角，它的存在将告知Spring容器需要为当前对象注入哪些依赖对象。而@Component则是配合Spring 2.5中新的classpath-scanning功能使用的。现在我们只要再向Spring的配置文件中增加一个“触发器”，使用@Autowired和@Component标注的类就能获得依赖对象的注入了。代码清单4-10给出的正是针对这部分功能的配置内容。

代码清单4-10　配置使用classpath-scanning功能

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" ➥
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ➥
xmlns:context="http://www.springframework.org/schema/context" ➥
xmlns:tx="http://www.springframework.org/schema/tx" ➥
xsi:schemaLocation="http://www.springframework.org/schema/beans ➥
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd ➥
http://www.springframework.org/schema/context ➥
http://www.springframework.org/schema/context/spring-context-2.5.xsd ➥
http://www.springframework.org/schema/tx ➥
http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">
<context:component-scan base-package="cn.spring21.project.base.package"/>
</beans>
```

<context:component-scan/>会到指定的包（package）下面扫描标注有@Component的类，如果找到，则将它们添加到容器进行管理，并根据它们所标注的@Autowired为这些类注入符合条件的依赖对象。

在以上所有这些工作都完成之后，我们就可以像通常那样加载配置并执行当前应用程序了，如以下代码所示：

```
public static void main(String[] args)
{
  ApplicationContext ctx = new ClassPathXmlApplicationContext("配置文件路径");
  FXNewsProvider newsProvider = (FXNewsProvider)container.getBean("FXNewsProvider");
```

```
    newsProvider.getAndPersistNews();
}
```

本章最后将详细讲解Spring 2.5新引入的“基于注解的依赖注入”。当前的内容只是让我们先从总体上有一个大概的印象，所以，不必强求自己现在就完全理解它们。

> **注意** Google Guice是一个完全基于注解方式、提供依赖注入服务的轻量级依赖注入框架，可以从Google Guice的站点获取有关这个框架的更多信息。

4.3 BeanFactory的XML之旅

XML格式的容器信息管理方式是Spring提供的最为强大、支持最为全面的方式。从Spring的参考文档到各Spring相关书籍，都是按照XML的配置进行说明的，这部分内容可以让你充分领略到Spring的IoC容器的魅力，以致于我们也不得不带你初次或者再次踏上Spring的XML之旅。

4.3.1 <beans>和<bean>

所有使用XML文件进行配置信息加载的Spring IoC容器，包括BeanFactory和ApplicationContext的所有XML相应实现，都使用统一的XML格式。在Spring 2.0版本之前，这种格式由Spring提供的DTD规定，也就是说，所有的Spring容器加载的XML配置文件的头部，都需要以下形式的DOCTYPE声明：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" ➥
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
   ...
</beans>
```

从Spring 2.0版本之后，Spring在继续保持向前兼容的前提下，既可以继续使用DTD方式的DOCTYPE进行配置文件格式的限定，又引入了基于XML Schema的文档声明。所以，Spring 2.0之后，同样可以使用代码清单4-11所展示的基于XSD的文档声明。

代码清单4-11 基于XSD的Spring配置文件文档声明

```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ➥
xmlns:util="http://www.springframework.org/schema/util" ➥
xmlns:jee="http://www.springframework.org/schema/jee" ➥
xmlns:lang="http://www.springframework.org/schema/lang" ➥
xmlns:aop="http://www.springframework.org/schema/aop" ➥
xmlns:tx="http://www.springframework.org/schema/tx" ➥
xsi:schemaLocation="http://www.springframework.org/schema/beans ➥
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd ➥
http://www.springframework.org/schema/util ➥
http://www.springframework.org/schema/util/spring-util-2.0.xsd ➥
http://www.springframework.org/schema/jee ➥
http://www.springframework.org/schema/jee/spring-jee-2.0.xsd ➥
http://www.springframework.org/schema/lang ➥
http://www.springframework.org/schema/lang/spring-lang-2.0.xsd ➥
http://www.springframework.org/schema/aop ➥
http://www.springframework.org/schema/aop/spring-aop-2.0.xsd ➥
```

```
http://www.springframework.org/schema/tx ➥
http://www.springframework.org/schema/tx/spring-tx-2.0.xsd">

</beans>
```

不过，不管使用哪一种形式的文档声明，实际上限定的元素基本上是相同的。让我们从最顶层的元素开始，看一下这两种文档声明都限定了哪些元素吧！

所有注册到容器的业务对象，在Spring中称之为Bean。所以，每一个对象在XML中的映射也自然而然地对应一个叫做<bean>的元素。既然容器最终可以管理所有的业务对象，那么在XML中把这些叫做<bean>的元素组织起来的，就叫做<beans>。多个<bean>组成一个<beans>很容易理解，不是吗？

1. <beans>之唯我独尊

<beans>是XML配置文件中最顶层的元素，它下面可以包含0或者1个<description>和多个<bean>以及<import>或者<alias>，如图4-4所示。

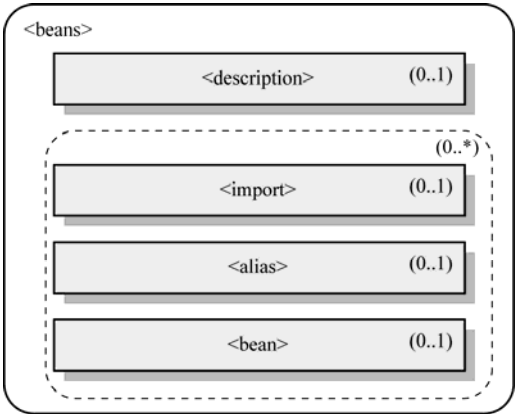

图4-4　<beans>与下一层元素的关系

<beans>作为所有<bean>的“统帅”，它拥有相应的属性（attribute）对所辖的<bean>进行统一的默认行为设置，包括如下几个。

- **default-lazy-init**。其值可以指定为true或者false，默认值为false。用来标志是否对所有的<bean>进行延迟初始化。
- **default-autowire**。可以取值为no、byName、byType、constructor以及autodetect。默认值为no，如果使用自动绑定的话，用来标志全体bean使用哪一种默认绑定方式。
- **default-dependency-check**。可以取值none、objects、simple以及all，默认值为none，即不做依赖检查。
- **default-init-method**。如果所管辖的<bean>按照某种规则，都有同样名称的初始化方法的话，可以在这里统一指定这个初始化方法名，而不用在每一个<bean>上都重复单独指定。
- **default-destroy-method**。与default-init-method相对应，如果所管辖的bean有按照某种规则使用了相同名称的对象销毁方法，可以通过这个属性统一指定。

注意　当然，如果你不清楚上面这些默认的属性具体有什么用，那也不必着急。在看完对<bean>的讲解之后，再回头来看，就会明了多了。给出这些信息，是想让你知道，如果在某

个场景下需要对大部分<bean>都重复设置某些行为的话，可以回头看一下，利用<beans>是否可以减少这种不必要的工作。

2. <description>、<import>和<alias>

之所以把这几个元素放到一起讲解，是因为通常情况下它们不是必需的。不过，了解一下也没什么不好，不是吗？

- <description>

可以通过<description>在配置的文件中指定一些描述性的信息。通常情况下，该元素是省略的。当然，如果愿意，<description>随时可以为我们效劳。

- <import>

通常情况下，可以根据模块功能或者层次关系，将配置信息分门别类地放到多个配置文件中。在想加载主要配置文件，并将主要配置文件所依赖的配置文件同时加载时，可以在这个主要的配置文件中通过<import>元素对其所依赖的配置文件进行引用。比如，如果A.xml中的<bean>定义可能依赖B.xml中的某些<bean>定义，那么就可以在A.xml中使用<import>将B.xml引入到A.xml，以类似于<import resource="B.xml"/>的形式。

但是，这个功能在我看来价值不大，因为容器实际上可以同时加载多个配置，没有必要非通过一个配置文件来加载所有配置。不过，或许在有些场景中使用这种方式比较方便也说不定。

- <alias>

可以通过<alias>为某些<bean>起一些“外号”（别名），通常情况下是为了减少输入。比如，假设有个<bean>，它的名称为dataSourceForMasterDatabase，你可以为其添加一个<alias>，像这样<alias name="dataSourceForMasterDatabase" alias="masterDataSource"/>。以后通过dataSourceForMasterDatabase或者masterDataSource来引用这个<bean>都可以，只要你觉得方便就行。

4.3.2 孤孤单单一个人

哦，错了，是孤孤单单一个Bean。每个业务对象作为个体，在Spring的XML配置文件中是与<bean>元素一一对应的。窥一斑而知全豹，只要我们了解单个的业务对象是如何配置的，剩下的就可以“依葫芦画瓢”了。所以，让我们先从最简单的单一对象配置开始吧！如下代码演示了最基础的对象配置形式：

```
<bean id="djNewsListener" class="..impl.DowJonesNewsListener">
</bean>
```

- id属性

通常，每个注册到容器的对象都需要一个唯一标志来将其与“同处一室”的“兄弟们”区分开来，就好像我们每一个人都有一个身份证号一样（重号的话就比较麻烦）。通过id属性来指定当前注册对象的beanName是什么。这里，通过id指定beanName为djNewsListener。实际上，并非任何情况下都需要指定每个<bean>的id，有些情况下，id可以省略，比如后面会提到的内部<bean>以及不需要根据beanName明确依赖关系的场合等。

除了可以使用id来指定<bean>在容器中的标志，还可以使用name属性来指定<bean>的别名（alias）。比如，以上定义，我们还可以像如下代码这样，为其添加别名：

```
<bean id="djNewsListener"
```

```
    name="/news/djNewsListener,dowJonesNewsListener"
    class="..impl.DowJonesNewsListener">
</bean>
```

与id属性相比，name属性的灵活之处在于，name可以使用id不能使用的一些字符，比如/。而且还可以通过逗号、空格或者冒号分割指定多个name。name的作用跟使用<alias>为id指定多个别名基本相同：

```
<alias name="djNewsListener" alias="/news/djNewsListener"/>
<alias name="djNewsListener" alias="dowJonesNewsListener"/>
```

- class属性

每个注册到容器的对象都需要通过<bean>元素的class属性指定其类型，否则，容器可不知道这个对象到底是何方神圣。在大部分情况下，该属性是必须的。仅在少数情况下不需要指定，如后面将提到的在使用抽象配置模板的情况下。

4.3.3　Help Me, Help You①

在大部分情况下，你不太可能选择单独"作战"，业务对象也是；各个业务对象之间会相互协作来更好地完成同一使命。这时，各个业务对象之间的相互依赖就是无法避免的。对象之间需要相互协作，在横向上它们存在一定的依赖性。而现在我们就是要看一下，在Spring的IoC容器的XML配置中，应该如何表达这种依赖性。

既然业务对象现在都符合IoC的规则，那么要了解的表达方式其实也很简单，无非就是看一下构造方法注入和setter方法注入通过XML是如何表达的而已。那么，让我们开始吧！

1. 构造方法注入的XML之道

按照Spring的IoC容器配置格式，要通过构造方法注入方式，为当前业务对象注入其所依赖的对象，需要使用<constructor-arg>。正常情况下，如以下代码所示：

```
<bean id="djNewsProvider" class="..FXNewsProvider">
  <constructor-arg>
    <ref bean="djNewsListener"/>
  </constructor-arg>
  <constructor-arg>
    <ref bean="djNewsPersister"/>
  </constructor-arg>
</bean>
```

对于<ref>元素，稍后会进行详细说明。这里你只需要知道，通过这个元素来指明容器将为djNewsProvider这个<bean>注入通过<ref>所引用的Bean实例。这种方式可能看起来或者编写起来不是很简洁，最新版本的Spring也支持配置简写形式，如以下代码所示：

```
<bean id="djNewsProvider" class="..FXNewsProvider">
  <constructor-arg ref="djNewsListener"/>
  <constructor-arg ref="djNewsPersister"/>
</bean>
```

简洁多了不是嘛？其实，无非就是表达方式不同而已，实际达到的效果是一样的。

有些时候，容器在加载XML配置的时候，因为某些原因，无法明确配置项与对象的构造方法参数列表的一一对应关系，就需要请<constructor-arg>的type或者index属性出马。比如，对象存在多

① 这句话是*Warcraft*中女巫的一句台词，这里用这句话来类比多个<bean>之间的关系：互相依赖，互相帮助以完成同一目标。

个构造方法，当参数列表数目相同而类型不同的时候，容器无法区分应该使用哪个构造方法来实例化对象，或者构造方法可能同时传入最少两个类型相同的对象。

- type属性

假设有一个对象定义如代码清单4-12所示。

代码清单4-12 随意声明的一个业务对象定义

```
public class MockBusinessObject {
  private String dependency1;
  private int    dependency2;

  public MockBusinessObject(String dependency)
  {
    this.dependency1 = dependency;
  }

  public MockBusinessObject(int dependency)
  {
    this.dependency2 = dependency;
  }
  ...

  @Override
  public String toString() {
    return new ToStringBuilder(this) ➥
    .append("dependency1", dependency1) ➥
    .append("dependency2", dependency2).toString();
  }
}
```

该类声明了两个构造方法，分别都只是传入一个参数，且参数类型不同。这时，我们可以进行配置，如以下代码所示：

```
<bean id="mockBO" class="..MockBusinessObject">
  <constructor-arg>
    <value>111111</value>
  </constructor-arg>
</bean>
```

如果从BeanFactory取得该对象并调用toString()查看的话，我们会发现Spring调用的是第一个构造方法，因为输出是如下内容：

```
..MockBusinessObject@f73c1[dependency1=111111,dependency2=0]
```

但是，如果我们想调用的却是第二个传入int类型参数的构造方法，又该如何呢？可以使用type属性，通过指定构造方法的参数类型来解决这一问题，配置内容如下代码所示：

```
<bean id="mockBO" class="..MockBusinessObject">
  <constructor-arg type="int">
    <value>111111</value>
  </constructor-arg>
</bean>
```

现在，我们得到了自己想要的对象实例，如下的控制台输出信息印证了这一点：

```
..MockBusinessObject@f73c1[dependency1=<null>,dependency2=111111]
```

- index属性

当某个业务对象的构造方法同时传入了多个类型相同的参数时，Spring又该如何将这些配置中的信息与实际对象的参数一一对应呢？好在，如果配置项信息和对象参数可以按照顺序初步对应的话，

Spring还是可以正常工作的，如代码清单4-13所示。

代码清单4-13　随意声明的一个业务对象定义

```
public class MockBusinessObject {
  private String dependency1;
  private String dependency2;

  public MockBusinessObject(String dependency1,String    dependency2)
  {
    this.dependency1 = dependency1;
    this.dependency2 = dependency2;
  }
  ...

  @Override
  public String toString() {
    return new ToStringBuilder(this) ➥
    .append("dependency1", dependency1) ➥
    .append("dependency2", dependency2).toString();
  }
}
```

并且，配置内容如以下代码所示：

```
<bean id="mockBO" class="..MockBusinessObject">
  <constructor-arg value="11111"/>
  <constructor-arg value="22222"/>
</bean>
```

那么，我们可以得到如下对象：

```
..MockBusinessObject@1ef8cf3[dependency1=11111,dependency2=22222]
```

但是，如果要让“11111”作为对象的第二个参数，而将“22222”作为第一个参数来构造对象，又该如何呢？好！可以颠倒配置项，如以下代码所示：

```
<bean id="mockBO" class="..MockBusinessObject">
  <constructor-arg value="22222"/>
  <constructor-arg value="11111"/>
</bean>
```

不过，还有一种方式，那就是像如下代码所示的那样，使用index属性：

```
<bean id="mockBO" class="..MockBusinessObject">
  <constructor-arg index="1" value="11111"/>
  <constructor-arg index="0" value="22222"/>
</bean>
```

这时，同样可以得到想要的对象实例，以下控制台输出表明了这一点：

```
..MockBusinessObject@ecd7e[dependency1=22222,dependency2=11111]
```

注意 index属性的取值从0开始，与一般的数组下标取值相同。所以，指定的第一个参数的index应该是0，第二个参数的index应该是1，依此类推。

2. setter方法注入的XML之道

与构造方法注入可以使用<constructor-arg>注入配置相对应，Spring为setter方法注入提供了<property>元素。

<property>有一个name属性（attribute），用来指定该<property>将会注入的对象所对应的实例变量名称。之后通过value或者ref属性或者内嵌的其他元素来指定具体的依赖对象引用或者值，如以下代码所示：

```
<bean id="djNewsProvider" class="..FXNewsProvider">
  <property name="newsListener">
    <ref bean="djNewsListener"/>
  </property>
  <property name="newPersistener">
    <ref bean="djNewsPersister"/>
  </property>
</bean>
```

当然，如果只是使用<property>进行依赖注入的话，请确保你的对象提供了默认的构造方法，也就是一个参数都没有的那个。

以上配置形式还可以简化为如下形式：

```
<bean id="djNewsProvider" class="..FXNewsProvider">
  <property name="newsListener" ref="djNewsListener"/>
  <property name="newPersistener" ref="djNewsPersister"/>
</bean>
```

使用<property>的setter方法注入和使用<constructor-arg>的构造方法注入并不是水火不容的。实际上，如果需要，可以同时使用这两个元素：

```
<bean id="mockBO" class="..MockBusinessObject">
  <constructor-arg value="11111"/>
  <property name="dependency2" value="22222"/>
</bean>
```

当然，现在需要MockBusinessObject提供一个只有一个String类型参数的构造方法，并且为dependency2提供了相应的setter方法。代码清单4-14演示了符合条件的一个业务对象定义。

代码清单4-14　随意声明的一个同时支持构造方法注入和setter方法注入的对象定义

```
public class MockBusinessObject {
  private String dependency1;
  private String dependency2;

  public MockBusinessObject(String dependency)
  {
    this.dependency1 = dependency;
  }

  public void setDependency2(String dependency2) {
    this.dependency2 = dependency2;
  }
  ...
}
```

3. <property>和<constructor-arg>中可用的配置项

之前我们看到，可以通过在<property>和<constructor-arg>这两个元素内部嵌套<value>或者<ref>，来指定将为当前对象注入的简单数据类型或者某个对象的引用。不过，为了能够指定多种注入类型，Spring还提供了其他的元素供我们使用，这包括bean、ref、idref、value、null、list、set、map、props。下面我们来逐个详细讲述它们。

提示　以下涉及的所有内嵌元素，对于<property>和<constructor-arg>都是通用的。

(1) <value>。可以通过value为主体对象注入简单的数据类型，不但可以指定String类型的数据，而且可以指定其他Java语言中的原始类型以及它们的包装器（wrapper）类型，比如int、Integer等。容器在注入的时候，会做适当的转换工作（我们会在后面揭示转换的奥秘）。你之前已经见过如何使用<value>了，不过让我们通过如下代码来重新认识一下它：

```
<constructor-arg>
   <value>111111</value>
</constructor-arg>
<property name="attributeName">
   <value>222222</value>
</property>
```

当然，如果愿意，你也可以使用如下的简化形式（不过这里的value是以上一层元素的属性身份出现）：

```
<constructor-arg value="111111"/>
<property name="attributeName" value="222222"/>
```

需要说明的是，<value>是最“底层”的元素，它内部不能再嵌套使用其他元素了。

(2) <ref>。使用ref来引用容器中其他的对象实例，可以通过ref的local、parent和bean属性来指定引用的对象的beanName是什么。代码清单4-15演示了ref及其三个对应属性的使用情况。

代码清单4-15　<ref>及其local、parent和bean属性的使用

```
constructor-arg>
   <ref local="djNewsPersister"/>
</constructor-arg>
或者
<constructor-arg>
   <ref parent="djNewsPersister"/>
</constructor-arg>
或者
<constructor-arg>
   <ref bean="djNewsPersister"/>
</constructor-arg>
```

local、parent和bean的区别在于：

- local只能指定与当前配置的对象在同一个配置文件的对象定义的名称（可以获得XML解析器的id约束验证支持）；
- parent则只能指定位于当前容器的父容器中定义的对象引用；

注意　BeanFactory可以分层次（通过实现HierarchicalBeanFactory接口），容器A在初始化的时候，可以首先加载容器B中的所有对象定义，然后再加载自身的对象定义，这样，容器B就成为了容器A的父容器，容器A可以引用容器B中的所有对象定义：

```
BeanFactory parentContainer = new XmlBeanFactory(new ClassPathResource("父容器配置文件路径"));
BeanFactory childContainer  = new XmlBeanFactory(new ClassPathResource("子容器配置文件路径"),parentContainer);
```

childContainer中定义的对象，如果通过parent指定依赖，则只能引用parentContainer中的对象定义。

- bean则基本上通吃，所以，通常情况下，直接使用bean来指定对象引用就可以了。

<ref>的定义为<!ELEMENT ref EMPTY>，也就是说，它下面没有其他子元素可用了，别硬往人家肚子里塞东西哦。

(3) <idref>。如果要为当前对象注入所依赖的对象的名称，而不是引用，那么通常情况下，可以使用<value>来达到这个目的，使用如下形式：

```
<property name="newsListenerBeanName">
  <value>djNewsListener</value>
</property>
```

但这种场合下，使用idref才是最为合适的。因为使用idref，容器在解析配置的时候就可以帮你检查这个beanName到底是否存在，而不用等到运行时才发现这个beanName对应的对象实例不存在。毕竟，输错名字的问题很常见。以下代码演示了idref的使用：

```
<property name="newsListenerBeanName">
  <idref bean="djNewsListener"/>
</property>
```

这段配置跟上面使用<value>达到了相同的目的，不过更加保险。如果愿意，也可以通过local而不是bean来指定最终值，不过，bean比较大众化哦。

(4) 内部<bean>。使用<ref>可以引用容器中独立定义的对象定义。但有时，可能我们所依赖的对象只有当前一个对象引用，或者某个对象定义我们不想其他对象通过<ref>引用到它。这时，我们可以使用内嵌的<bean>，将这个私有的对象定义仅局限在当前对象。对于FX新闻系统的DowJonesNewsListener而言，实际上只有道琼斯的FXNewsProvider会使用它。而且，我们也不想让其他对象引用到它。为此完全可以像代码清单4-16这样，将它配置为内部<bean>的形式。

代码清单4-16　内部<bean>的配置演示

```
<bean id="djNewsProvider" class="..FXNewsProvider">
  <constructor-arg index="0">
    <bean class="..impl.DowJonesNewsListener">
    </bean>
  </constructor-arg>
  <constructor-arg index="1">
    <ref bean="djNewsPersister"/>
  </constructor-arg>
</bean>
```

这样，该对象实例就只有当前的djNewsProvider可以使用，其他对象无法取得该对象的引用。

> **注意**　因为就只有当前对象引用内部<bean>所指定的对象，所以，内部<bean>的id不是必须的。当然，如果你愿意指定id，那也是无所谓的。如下所示：
>
> ```
> <constructor-arg index="0">
>   <bean id="djNewsListener" class="..impl.DowJonesNewsListener">
>   </bean>
> </constructor-arg>
> ```

内部<bean>的配置只是在位置上有所差异，但配置项上与其他的<bean>是没有任何差别的。也就是说，<bean>内嵌的所有元素，内部<bean>的<bean>同样可以使用。如果内部<bean>对应的对象还依赖于其他对象，你完全可以像其他独立的<bean>定义一样为其配置相关依赖，没有任何差别。

(5) <list>。<list>对应注入对象类型为java.util.List及其子类或者数组类型的依赖对象。

通过<list>可以有序地为当前对象注入以collection形式声明的依赖。代码清单4-17给出了一个使用<list>的实例演示。

代码清单4-17　使用<list>进行依赖注入的对象定义以及相关配置内容

```
public class MockDemoObject
{
  private List param1;
  private String[] param2;
  ...
  // 相应的setter和getter方法
  ...
}

配置类似于
<property name="param1">
  <list>
    <value> something</value>
    <ref bean="someBeanName"/>
    <bean class="..."/>
  </list>
</property>
<property name="param2">
  <list>
    <value>stringValue1</value>
    <value>stringValue2</value>
  </list>
</property>
```

注意，<list>元素内部可以嵌套其他元素，并且可以像param1所展示的那样夹杂配置。但是，从好的编程实践来说，这样的处理并不恰当，除非你真的知道自己在做什么！（以上只是出于演示的目的才会如此配置）。

(6) <set>。如果说<list>可以帮你有序地注入一系列依赖的话，那么<set>就是无序的，而且，对于set来说，元素的顺序本来就是无关紧要的。<set>对应注入Java Collection中类型为java.util.Set或者其子类的依赖对象。代码清单4-18演示了通常情况下<set>的使用场景。

代码清单4-18　使用<set>进行依赖注入的对象定义以及相关配置内容

```
public class MockDemoObject
{
  private Set valueSet;
  // 必要的setter和getter方法
  ...
}
配置类似于
<property name="valueSet">
  <set>
    <value> something</value>
    <ref bean="someBeanName"/>
    <bean class="..."/>
    <list>
      ...
    </list>
  </set>
</property>
```

例子就是例子，只是为了给你演示这个元素到底有多少能耐。从配置上来说，这样多层嵌套、多

元素混杂配置是完全没有问题的。不过，各位在具体编程实践的时候可要小心了。如果你真的想这么夹杂配置的话，不好意思，估计ClassCastException会很愿意来亲近你，而这跟容器或者配置一点儿关系也没有。

(7) <map>。与列表（list）使用数字下标来标识元素不同，映射（map）可以通过指定的键（key）来获取相应的值。如果说在<list>中混杂不同元素不是一个好的实践（方式）的话，你就应该求助<map>。<map>与<list>和<set>的相同点在于，都是为主体对象注入Collection类型的依赖，不同点在于它对应注入java.util.Map或者其子类类型的依赖对象。代码清单4-19演示了<map>的通常使用场景。

代码清单4-19 使用<map>进行依赖注入的对象定义以及相关配置内容

```
public class MockDemoObject
{
  private Map mapping;
  // 必要的setter和getter方法
  ...
}
配置类似于
<property name="mapping">
  <map>
    <entry key="strValueKey">
      <value>something</value>
    </entry>
    <entry>
      <key>objectKey</key>
      <ref bean="someObject"/>
    </entry>
    <entry key-ref="lstKey">
      <list>
        ...
      </list>
    </entry>
    ...
  </map>
</property>
```

对于<map>来说，它可以内嵌任意多个<entry>，每一个<entry>都需要为其指定一个键和一个值，就跟真正的java.util.Map所要求的一样。

- 指定entry的键。可以使用<entry>的属性——key或者key-ref来指定键，也可以使用<entry>的内嵌元素<key>来指定键，这完全看个人喜好，但两种方式可以达到相同的效果。在<key>内部，可以使用以上提到的任何元素来指定键，从简单的<value>到复杂的Collection，只要映射需要，你可以任意发挥。
- 指定entry对应的值。<entry>内部可以使用的元素，除了<key>是用来指定键的，其他元素可以任意使用，来指定entry对应的值。除了之前提到的那些元素，还包括马上就要谈到的<props>。如果对应的值只是简单的原始类型或者单一的对象引用，也可以直接使用<entry>的value或者value-ref这两个属性来指定，从而省却多敲入几个字符的工作量。

> **注意** key属性用于指定通常的简单类型的键，而key-ref则用于指定对象的引用作为键。

所以，如果你不想敲那么些字符，可以像代码清单4-20所展示的那样使用<map>进行依赖注入的

配置。

代码清单4-20　简化版的<map>配置使用演示

```
public class MockDemoObject
{
   private Map mapping;
   // 必要的setter和getter方法
   ...
}
配置类似于
<property name="valueSet">
   <map>
      <entry key="strValueKey" value="something"/>
      <entry key-ref="" value-ref="someObject"/>
      <entry key-ref="lstKey">
         <list>
            ...
         </list>
      </entry>
      ...
   </map>
</property>
```

(8) <props>。<props>是简化后了的<map>，或者说是特殊化的map，该元素对应配置类型为java.util.Properties的对象依赖。因为Properties只能指定String类型的键（key）和值，所以，<props>的配置简化很多，只有固定的格式，见代码清单4-21。

代码清单4-21　使用<props>进行依赖注入的场景演示

```
public class MockDemoObject
{
   private Properties emailAddrs;
   // 必要的setter和getter方法
   ...
}
配置类似于
<property name="valueSet">
   <props>
      <prop key="author">fujohnwang@gmail.com</prop>
      <prop key="support">support@spring21.cn</prop>
      ...
   </props>
</property>
```

每个<props>可以嵌套多个<prop>，每个<prop>通过其key属性来指定键，在<prop>内部直接指定其所对应的值。<prop>内部没有任何元素可以使用，只能指定字符串，这个是由java.util.Properties的语意决定的。

(9) <null/>。最后一个提到的元素是<null/>，这是最简单的一个元素，因为它只是一个空元素，而且通常使用到它的场景也不是很多。对于String类型来说，如果通过value以这样的方式指定注入，即<value></value>，那么，得到的结果是""，而不是null。所以，如果需要为这个string对应的值注入null的话，请使用<null/>。当然，这并非仅限于String类型，如果某个对象也有类似需求，请不要犹豫。代码清单4-22演示了一个使用<null/>的简单场景。

代码清单4-22 使用`<null/>`进行依赖注入的简单场景演示

```
public class MockDemoObject
{
  private String param1;
  private Object param2;
  // 必要的setter和getter方法
  ...
}
配置为
<property name="param1">
  <null/>
</property>
<property name="param2">
  <null/>
</property>
实际上就相当于
public class MockDemoObject
{
  private String param1=null;
  private Object param2=null;
  // 必要的setter和getter方法
  ...
}
虽然这里看起来没有太大意义!
```

4. depends-on

通常情况下，可以直接通过之前提到的所有元素，来显式地指定bean之间的依赖关系。这样，容器在初始化当前bean定义的时候，会根据这些元素所标记的依赖关系，首先实例化当前bean定义所依赖的其他bean定义。但是，如果某些时候，我们没有通过类似`<ref>`的元素明确指定对象A依赖于对象B的话，如何让容器在实例化对象A之前首先实例化对象B呢？

考虑以下所示代码：

```
public class SystemConfigurationSetup
{
  static
  {
    DOMConfigurator.configure("配置文件路径");
    // 其他初始化代码
  }
  ...
}
```

系统中所有需要日志记录的类，都需要在这些类使用之前首先初始化log4j。那么，就会非显式地依赖于`SystemConfigurationSetup`的静态初始化块。如果`ClassA`需要使用log4j，那么就必须在bean定义中使用`depends-on`来要求容器在初始化自身实例之前首先实例化`SystemConfigurationSetup`，以保证日志系统的可用，如下代码演示的正是这种情况：

```
<bean id="classAInstance" class="...ClassA" depends-on="configSetup"/>

<bean id="configSetup" class="SystemConfigurationSetup"/>
```

举log4j在静态代码块（static block）中初始化的例子在实际系统中其实不是很合适，因为通常在应用程序的主入口类初始化日志就可以了。这里主要是给出`depends-on`可能的使用场景，大部分情况下，是那些拥有静态代码块初始化代码或者数据库驱动注册之类的场景。

如果说ClassA拥有多个类似的非显式依赖关系，那么，你可以在ClassA的depends-on中通过逗号分割各个beanName，如下代码所示：

```
<bean id="classAInstance" class="...ClassA" depends-on="configSetup,configSetup2,..."/>

<bean id="configSetup" class="SystemConfigurationSetup"/>
<bean id="configSetup2" class="SystemConfigurationSetup2"/>
```

5. **autowire**

除了可以通过配置明确指定bean之间的依赖关系，Spirng还提供了根据bean定义的某些特点将相互依赖的某些bean直接自动绑定的功能。通过<bean>的autowire属性，可以指定当前bean定义采用某种类型的自动绑定模式。这样，你就无需手工明确指定该bean定义相关的依赖关系，从而也可以免去一些手工输入的工作量。

Spring提供了5种自动绑定模式，即no、byName、byType、constructor和autodetect，下面是它们的具体介绍。

- no

容器默认的自动绑定模式，也就是不采用任何形式的自动绑定，完全依赖手工明确配置各个bean之间的依赖关系，以下代码演示的两种配置是等效的：

```
<bean id="beanName" class="..."/>

或者

<bean id="beanName" class="..." autowire="no"/>
```

- byName

按照类中声明的实例变量的名称，与XML配置文件中声明的bean定义的beanName的值进行匹配，相匹配的bean定义将被自动绑定到当前实例变量上。这种方式对类定义和配置的bean定义有一定的限制。假设我们有如下所示的类定义：

```
public class Foo
{
  private Bar emphasisAttribute;
  ...
  // 相应的setter方法定义
}
public class Bar
{
  ...
}
```

那么应该使用如下代码所演示的自动绑定定义，才能达到预期的目的：

```
<bean id="fooBean" class="...Foo" autowire="byName">
</bean>
<bean id="emphasisAttribute" class="...Bar">
</bean>
```

需要注意两点：第一，我们并没有明确指定fooBean的依赖关系，而仅指定了它的autowire属性为byName；第二，第二个bean定义的id为emphasisAttribute，与Foo类中的实例变量名称相同。

- byType

如果指定当前bean定义的autowire模式为byType，那么，容器会根据当前bean定义类型，分析其相应的依赖对象类型，然后到容器所管理的所有bean定义中寻找与依赖对象类型相同的bean定义，然

后将找到的符合条件的bean自动绑定到当前bean定义。

对于byName模式中的实例类Foo来说，容器会在其所管理的所有bean定义中寻找类型为Bar的bean定义。如果找到，则将找到的bean绑定到Foo的bean定义；如果没有找到，则不做设置。但如果找到多个，容器会告诉你它解决不了“该选用哪一个”的问题，你只好自己查找原因，并自己修正该问题。所以，byType只能保证，在容器中只存在一个符合条件的依赖对象的时候才会发挥最大的作用，如果容器中存在多个相同类型的bean定义，那么，不好意思，采用手动明确配置吧！

指定byType类型的autowire模式与byName没什么差别，只是autowire的值换成byType而已，可以参考如下代码：

```
<bean id="fooBean" class="...Foo" autowire="byType">
</bean>

<bean id="anyName" class="...Bar">
</bean>
```

- constructor

byName和byType类型的自动绑定模式是针对property的自动绑定，而constructor类型则是针对构造方法参数的类型而进行的自动绑定，它同样是byType类型的绑定模式。不过，constructor是匹配构造方法的参数类型，而不是实例属性的类型。与byType模式类似，如果找到不止一个符合条件的bean定义，那么，容器会返回错误。使用上也与byType没有太大差别，只不过是应用到需要使用构造方法注入的bean定义之上，代码清单4-23给出了一个使用construtor模式进行自动绑定的简单场景演示。

代码清单4-23　construtor类型自动绑定的使用场景演示

```
public class Foo
{
  private Bar bar;
  public Foo(Bar arg)
  {
    this.bar = arg;
  }
  ...
}
相应配置为
<bean id="foo" class="...Foo" autowire="constructor"/>

<bean id="bar" class="...Bar">
</bean>
```

- autodetect

这种模式是byType和constructor模式的结合体，如果对象拥有默认无参数的构造方法，容器会优先考虑byType的自动绑定模式。否则，会使用constructor模式。当然，如果通过构造方法注入绑定后还有其他属性没有绑定，容器也会使用byType对剩余的对象属性进行自动绑定。

小心

- 手工明确指定的绑定关系总会覆盖自动绑定模式的行为。
- 自动绑定只应用于“原生类型、String类型以及Classes类型以外”的对象类型，对“原生类型、String类型和Classes类型”以及“这些类型的数组”应用自动绑定是无效的。

自动绑定与手动明确绑定

自动绑定和手动明确绑定各有利弊。自动绑定的优点有如下两点。

(1) 某种程度上可以有效减少手动敲入配置信息的工作量。

(2) 某些情况下，即使为当前对象增加了新的依赖关系，但只要容器中存在相应的依赖对象，就不需要更改任何配置信息。

自动绑定的缺点有如下几点。

(1) 自动绑定不如明确依赖关系一目了然。我们可以根据明确的依赖关系对整个系统有一个明确的认识，但使用自动绑定的话，就可能需要在类定义以及配置文件之间，甚至各个配置文件之间来回转换以取得相应的信息。

(2) 某些情况下，自动绑定无法满足系统需要，甚至导致系统行为异常或者不可预知。根据类型（byType）匹配进行的自动绑定，如果系统中增加了另一个相同类型的bean定义，那么整个系统就会崩溃；根据名字（byName）匹配进行的自动绑定，如果把原来系统中相同名称的bean定义类型给换掉，就会造成问题，而这些可能都是在不经意间发生的。

(3) 使用自动绑定，我们可能无法获得某些工具的良好支持，比如Spring IDE。

通常情况下，只要有良好的XML编辑器支持，我不会介意多敲那几个字符。起码自己可以对整个系统的行为有完全的把握。当然，任何事物都不绝对，只要根据相应场景找到合适的就可以。

噢，对了，差点儿忘了！作为所有<bean>的统帅，<beans>有一个default-autowire属性，它可以帮我们省去为多个<bean>单独设置autowire属性的麻烦，default-autowire的默认值为no，即不进行自动绑定。如果想让系统中所有的<bean>定义都使用byType模式的自动绑定，我们可以使用如下配置内容：

```
<beans default-autowire="byType">
  <bean id="..." class="..."/>
  ...
</beans>
```

6. **dependency-check**

我们可以使用每个<bean>的dependency-check属性对其所依赖的对象进行最终检查，就好像电影里每队美国大兵上战场之前，带队的军官都会朝着士兵大喊“检查装备，check，recheck”是一个道理。该功能主要与自动绑定结合使用，可以保证当自动绑定完成后，最终确认每个对象所依赖的对象是否按照所预期的那样被注入。当然，并不是说不可以与平常的明确绑定方式一起使用。

该功能可以帮我们检查每个对象某种类型的所有依赖是否全部已经注入完成，不过可能无法细化到具体的类型检查。但某些时候，使用setter方法注入就是为了拥有某种可以设置也可以不设置的灵活性，所以，这种依赖检查并非十分有用，尤其是在手动明确绑定依赖关系的情况下。

与军官会让大兵检查枪支弹药和防弹衣等不同装备一样，可以通过dependency-check指定容器帮我们检查某种类型的依赖，基本上有如下4种类型的依赖检查。

- none。不做依赖检查。将dependency-check指定为none跟不指定这个属性等效，所以，还是不要多敲那几个字符了吧。默认情况下，容器以此为默认值。
- simple。如果将dependency-check的值指定为simple，那么容器会对简单属性类型以及相关的collection进行依赖检查，对象引用类型的依赖除外。
- object。只对对象引用类型依赖进行检查。

- all。将simple和object相结合，也就是说会对简单属性类型以及相应的collection和所有对象引用类型的依赖进行检查。

总地来说，控制得力的话，这个依赖检查的功能我们基本可以不考虑使用。

7. **lazy-init**

延迟初始化（lazy-init）这个特性的作用，主要是可以针对ApplicationContext容器的bean初始化行为施以更多控制。与BeanFactory不同，ApplicationContext在容器启动的时候，就会马上对所有的“singleton的bean定义”[①]进行实例化操作。通常这种默认行为是好的，因为如果系统有问题的话，可以在第一时间发现这些问题，但有时，我们不想某些bean定义在容器启动后就直接实例化，可能出于容器启动时间的考虑，也可能出于其他原因的考虑。总之，我们想改变某个或者某些bean定义在ApplicationContext容器中的默认实例化时机。这时，就可以通过<bean>的lazy-init属性来控制这种初始化行为，如下代码所示：

```
<bean id="lazy-init-bean" class="..." lazy-init="true"/>
<bean id="not-lazy-init-bean" class="..."/>
```

这样，ApplicationContext容器在启动的时候，只会默认实例化not-lazy-init-bean而不会实例化lazy-init-bean。

当然，仅指定lazy-init-bean的lazy-init为true，并不意味着容器就一定会延迟初始化该bean的实例。如果某个非延迟初始化的bean定义依赖于lazy-init-bean，那么毫无疑问，按照依赖决计的顺序，容器还是会首先实例化lazy-init-bean，然后再实例化后者，如下代码演示了这种相互牵连导致延迟初始化失败的情况：

```
<bean id="lazy-init-bean" class="..." lazy-init="true"/>

<bean id="not-lazy-init-bean" class="...">
  <property name="propName">
    <ref bean="lazy-init-bean"/>
  </property>
</bean>
```

虽然lazy-init-bean是延迟初始化的，但因为依赖它的not-lazy-init-bean并不是延迟初始化，所以lazy-init-bean还是会被提前初始化，延迟初始化的良好打算“泡汤”。如果我们真想保证lazy-init-bean一定会被延迟初始化的话，就需要保证依赖于该bean定义的其他bean定义也同样设置为延迟初始化。在bean定义很多时，好像工作量也不小哦。不过不要忘了，<beans>可是所有<bean>的统领啊，让它一声令下吧！如代码清单4-24所演示的，在顶层由<beans>统一控制延迟初始化行为即可。

代码清单4-24　通过<beans>设置统一的延迟初始化行为

```
<beans default-lazy-init="true">
  <bean id="lazy-init-bean" class="..."/>

  <bean id="not-lazy-init-bean" class="...">
    <property name="propName">
      <ref bean="lazy-init-bean"/>
    </property>
  </bean>
  ...
</beans>
```

① 对于singleton的概念，参考4.3.5节。

这样我们就不用每个<bean>都设置一遍，省事儿多了不是吗？

4.3.4　继承？我也会！

除了单独存在的bean以及多个bean之间的横向依赖关系，我们也不能忽略“纵向上”各个bean之间的关系。确切来讲，我其实是想说“类之间的继承关系”。不可否认，继承可是在面向对象界声名远扬啊。

假设我们某一天真的需要对FXNewsProvider使用继承进行扩展，那么可能会声明如下代码所示的子类定义：

```
class SpecificFXNewsProvider extends FXNewsProvider
{
  private IFXNewsListener  newsListener;
  private IFXNewsPersister newPersistener;
  ...
}
```

实际上，我们想让该子类与FXNewsProvider使用相同的IFXNewsPersister，即DowJonesNews-Persister，那么可以使用如代码清单4-25所示的配置。

代码清单4-25　使用同一个IFXNewsPersister依赖对象的FXNewsProvider和SpecificFXNews-Provider配置内容

```
<bean id="superNewsProvider" class="..FXNewsProvider">
  <property name="newsListener">
    <ref bean="djNewsListener"/>
  </property>
  <property name="newPersistener">
    <ref bean="djNewsPersister"/>
  </property>
</bean>

<bean id="subNewsProvider" class="..SpecificFXNewsProvider">
  <property name="newsListener">
    <ref bean="specificNewsListener"/>
  </property>
  <property name="newPersistener">
    <ref bean="djNewsPersister"/>
  </property>
</bean>
```

但实际上，这种配置存在冗余，而且也没有表现两者之间的纵向关系。所以，我们可以引入XML中的bean的“继承”配置，见代码清单4-26。

代码清单4-26　使用继承关系配置的FXNewsProvider和SpecificFXNewsProvider

```
<bean id="superNewsProvider" class="..FXNewsProvider">
  <property name="newsListener">
    <ref bean="djNewsListener"/>
  </property>
  <property name="newPersistener">
    <ref bean="djNewsPersister"/>
  </property>
</bean>

<bean id="subNewsProvider" parent="superNewsProvider"
  class="..SpecificFXNewsProvider">
  <property name="newsListener">
```

```
        <ref bean="specificNewsListener"/>
    </property>
</bean>
```

我们在声明subNewsProvider的时候，使用了parent属性，将其值指定为superNewsProvider，这样就继承了superNewsProvider定义的默认值，只需要将特定的属性进行更改，而不要全部又重新定义一遍。

parent属性还可以与abstract属性结合使用，达到将相应bean定义模板化的目的。比如，我们还可以像代码清单4-27所演示的这样声明以上类定义。

代码清单4-27 使用模板化配置形式配置FXNewsProvider和SpecificFXNewsProvider

```
<bean id="newsProviderTemplate" abstract="true">
    <property name="newPersistener">
        <ref bean="djNewsPersister"/>
    </property>
</bean>

<bean id="superNewsProvider" parent="newsProviderTemplate"
    class="..FXNewsProvider">
    <property name="newsListener">
        <ref bean="djNewsListener"/>
    </property>
</bean>

<bean id="subNewsProvider" parent="newsProviderTemplate"
    class="..SpecificFXNewsProvider">
    <property name="newsListener">
        <ref bean="specificNewsListener"/>
    </property>
</bean>
```

newsProviderTemplate的bean定义通过abstract属性声明为true，说明这个bean定义不需要实例化。实际上，这就是之前提到的可以不指定class属性的少数场景之一（当然，同时指定class和abstract="true"也是可以的）。该bean定义只是一个配置模板，不对应任何对象。superNewsProvider和subNewsProvider通过parent指向这个模板定义，就拥有了该模板定义的所有属性配置。当多个bean定义拥有多个相同默认属性配置的时候，你会发现这种方式可以带来很大的便利。

另外，既然这里提到abstract，对它就多说几句。容器在初始化对象实例的时候，不会关注将abstract属性声明为true的bean定义。如果你不想容器在初始化的时候实例化某些对象，那么可以将其abstract属性赋值true，以避免容器将其实例化。对于ApplicationContext容器尤其如此，因为默认情况下，ApplicationContext会在容器启动的时候就对其管理的所有bean进行实例化，只有标志为abstract的bean除外。

4.3.5 bean 的 scope

BeanFactory除了拥有作为IoC Service Provider的职责，作为一个轻量级容器，它还有着其他一些职责，其中就包括对象的生命周期管理。

本节主要讲述容器中管理的对象的scope这个概念。多数中文资料在讲解bean的scope时喜欢用“作用域”这个名词，应该还算贴切吧。不过，我更希望告诉你scope这个词到底代表什么意思，至于你怎么称呼它反而不重要。

scope用来声明容器中的对象所应该处的限定场景或者说该对象的存活时间，即容器在对象进入其相应的scope之前，生成并装配这些对象，在该对象不再处于这些scope的限定之后，容器通常会销毁

这些对象。打个比方吧！我们都是处于社会（容器）中，如果把中学教师作为一个类定义，那么当容器初始化这些类之后，中学教师只能局限在中学这样的场景中；中学，就可以看作中学教师的scope。

Spring容器最初提供了两种bean的scope类型：singleton和prototype，但发布2.0之后，又引入了另外三种scope类型，即request、session和global session类型。不过这三种类型有所限制，只能在Web应用中使用。也就是说，只有在支持Web应用的ApplicationContext中使用这三个scope才是合理的。

我们可以通过使用<bean>的singleton或者scope属性来指定相应对象的scope，其中，scope属性只能在XSD格式的文档声明中使用，类似于如下代码所演示的形式：

```
DTD:
<bean id="mockObject1" class="...MockBusinessObject" singleton="false"/>
XSD:
<bean id="mockObject2" class="...MockBusinessObject" scope="prototype"/>
```

让我们来看一下容器提供的这几个scope是如何限定相应对象的吧！

1. singleton

配置中的bean定义可以看作是一个模板，容器会根据这个模板来构造对象。但是要根据这个模板构造多少对象实例，又该让这些构造完的对象实例存活多久，则由容器根据bean定义的scope语意来决定。标记为拥有singleton scope的对象定义，在Spring的IoC容器中只存在一个实例，所有对该对象的引用将共享这个实例。该实例从容器启动，并因为第一次被请求而初始化之后，将一直存活到容器退出，也就是说，它与IoC容器“几乎”拥有相同的“寿命”。

图4-5是Spring参考文档中所给出的singleton的bean的实例化和注入语意演示图例，或许可以更形象地说明问题。

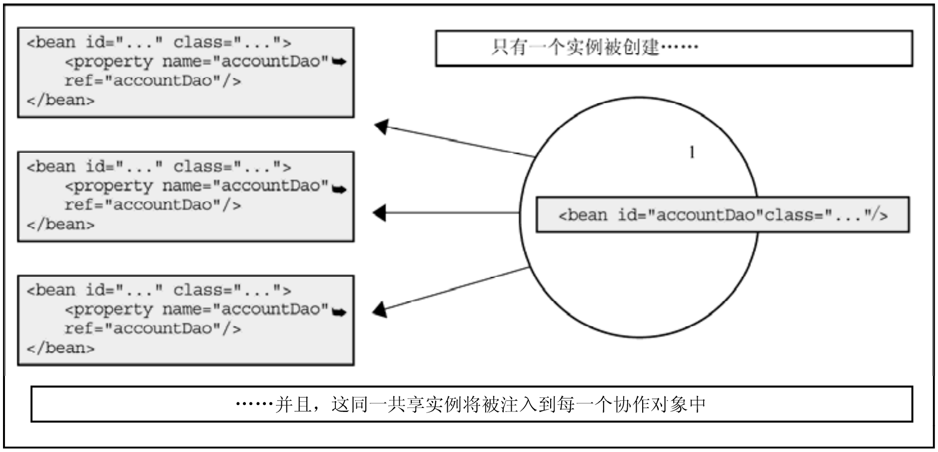

图4-5　singleton scope

需要注意的一点是，不要因为名字的原因而与GoF[1]所提出的Singleton模式相混淆，二者的语意是不同的：标记为singleton的bean是由容器来保证这种类型的bean在同一个容器中只存在一个共享实例；而Singleton模式则是保证在同一个Classloader中只存在一个这种类型的实例。

可以从两个方面来看待singleton的bean所具有的特性。

- **对象实例数量**。singleton类型的bean定义，在一个容器中只存在一个共享实例，所有对该类型

① Gang of Four，*Design Patterns: Elements of Reusable Object Software*一书的四位作者的称号。

bean的依赖都引用这一单一实例。这就好像每个幼儿园都会有一个滑梯一样，这个幼儿园的小朋友共同使用这一个滑梯。而对于该幼儿园容器来说，滑梯实际上就是一个singleton的bean。

- **对象存活时间**。singleton类型bean定义，从容器启动，到它第一次被请求而实例化开始，只要容器不销毁或者退出，该类型bean的单一实例就会一直存活。

通常情况下，如果你不指定bean的scope，singleton便是容器默认的scope，所以，下面三种配置形式实际上达成的是同样的效果：

```
<!-- DTD or XSD -->
<bean id="mockObject1" class="...MockBusinessObject"/>
<!-- DTD -->
<bean id="mockObject1" class="...MockBusinessObject" singleton="true"/>
<!-- XSD -->
<bean id="mockObject1" class="...MockBusinessObject" scope="singleton"/>
```

2. prototype

针对声明为拥有prototype scope的bean定义，容器在接到该类型对象的请求的时候，会每次都重新生成一个新的对象实例给请求方。虽然这种类型的对象的实例化以及属性设置等工作都是由容器负责的，但是只要准备完毕，并且对象实例返回给请求方之后，容器就不再拥有当前返回对象的引用，请求方需要自己负责当前返回对象的后继生命周期的管理工作，包括该对象的销毁。也就是说，容器每次返回给请求方一个新的对象实例之后，就任由这个对象实例“自生自灭”了。

让我们继续幼儿园的比喻，看看prototype在这里应该映射到哪些事物。儿歌里好像有句“排排坐，分果果”，我们今天要分苹果咯！将苹果的bean定义的scope声明为prototype，在每个小朋友领取苹果的时候，我们都是分发一个新的苹果给他。发完之后，小朋友爱怎么吃怎么吃，爱什么时候吃什么时候吃。但是，吃完后要记得把果核扔到果皮箱哦！ 而如果你把苹果的bean定义的scope声明为singleton会是什么情况呢？如果第一个小朋友比较谦让，那么他可能对这个苹果只咬一口，但是下一个小朋友吃多少就不知道了。当吃得只剩一个果核的时候，下一个来吃苹果的小朋友肯定要哭鼻子的。

所以，对于那些请求方不能共享使用的对象类型，应该将其bean定义的scope设置为prototype。这样，每个请求方可以得到自己对应的一个对象实例，而不会出现上面“哭鼻子”的现象。通常，声明为prototype的scope的bean定义类型，都是一些有状态的，比如保存每个顾客信息的对象。

从Spring 参考文档上的这幅图片（见图4-6），你可以再次了解一下拥有prototype scope的bean定义，在实例化对象并注入依赖的时候，它的具体语意是个什么样子。

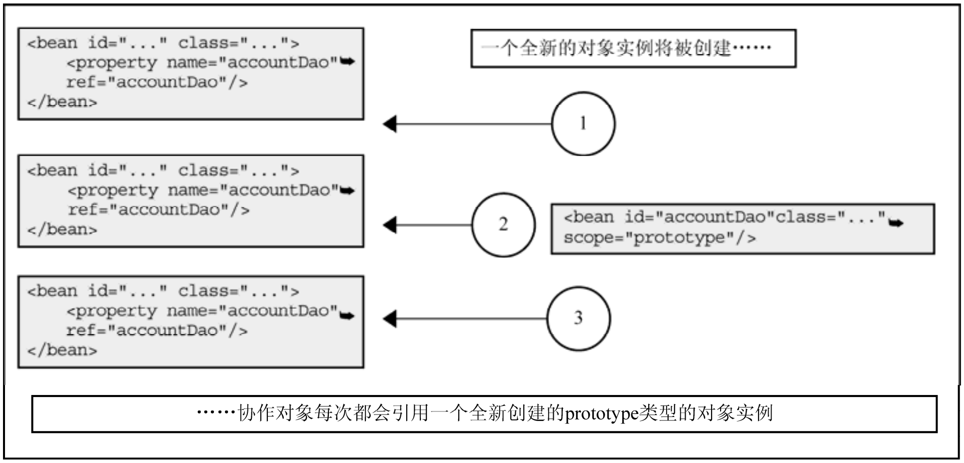

图4-6 prototype scope

你用以下形式来指定某个bean定义的scope为prototype类型，效果是一样的：

```
<!-- DTD -->
<bean id="mockObject1" class="...MockBusinessObject" singleton="false"/>
<!-- XSD -->
<bean id="mockObject1" class="...MockBusinessObject" scope="prototype"/>
```

3. request、session和global session

这三个scope类型是Spirng 2.0之后新增加的，它们不像之前的singleton和prototype那么“通用”，因为它们只适用于Web应用程序，通常是与XmlWebApplicationContext共同使用，而这些将在第6部分详细讨论。不过，既然它们也属于scope的概念，这里就简单提几句。

> **注意** 只能使用scope 属性才能指定这三种“bean的scope类型”。也就是说，你不得不使用基于XSD文档声明的XML配置文件格式。

- request

request通常的配置形式如下：

```
<bean id="requestProcessor" class="...RequestProcessor" scope="request"/>
```

Spring容器，即XmlWebApplicationContext会为每个HTTP请求创建一个全新的Request-Processor对象供当前请求使用，当请求结束后，该对象实例的生命周期即告结束。当同时有10个HTTP请求进来的时候，容器会分别针对这10个请求返回10个全新的RequestProcessor对象实例，且它们之间互不干扰。从不是很严格的意义上说，request可以看作prototype的一种特例，除了场景更加具体之外，语意上差不多。

- session

对于Web应用来说，放到session中的最普遍的信息就是用户的登录信息，对于这种放到session中的信息，我们可使用如下形式指定其scope为session：

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="session"/>
```

Spring容器会为每个独立的session创建属于它们自己的全新的UserPreferences对象实例。与

request相比，除了拥有session scope的bean的实例具有比request scope的bean可能更长的存活时间，其他方面真是没什么差别。

- global session

还是userPreferences，不过scope对应的值换一下，如下所示：

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="globalSession"/>
```

global session只有应用在基于portlet的Web应用程序中才有意义，它映射到portlet的global范围的session。如果在普通的基于servlet的Web应用中使用了这个类型的scope，容器会将其作为普通的session类型的scope对待。

4. 自定义scope类型

在Spring 2.0之后的版本中，容器提供了对scope的扩展点，这样，你可以根据自己的需要或者应用的场景，来添加自定义的scope类型。需要说明的是，默认的singleton和prototype是硬编码到代码中的，而request、session和global session，包括自定义scope类型，则属于可扩展的scope行列，它们都实现了org.springframework.beans.factory.config.Scope接口，该接口定义如下：

```
public interface Scope {

  Object get(String name, ObjectFactory objectFactory);

  Object remove(String name);

  void registerDestructionCallback(String name, Runnable callback);

  String getConversationId();
}
```

要实现自己的scope类型，首先需要给出一个Scope接口的实现类，接口定义中的4个方法并非都是必须的，但get和remove方法必须实现。我们可以看一下http://www.jroller.com/eu/entry/implementing_efficinet_id_generator中提到的一个ThreadScope的实现（见代码清单4-28）。

代码清单4-28　自定义的ThreadScope的定义

```
public class ThreadScope implements Scope {

  private final ThreadLocal threadScope = new ThreadLocal() {
      protected Object initialValue() {
        return new HashMap();
      }
    };

  public Object get(String name, ObjectFactory objectFactory) {
    Map scope = (Map) threadScope.get();
    Object object = scope.get(name);
    if(object==null) {
      object = objectFactory.getObject();
      scope.put(name, object);
    }
    return object;
  }

  public Object remove(String name) {
    Map scope = (Map) threadScope.get();
    return scope.remove(name);
  }
```

```
    public void registerDestructionCallback(String name, Runnable callback) {
    }
    ...
}
```

更多Scope相关的实例，可以参照同一站点的一篇文章“More fun with Spring scopes”（http://jroller.com/eu/entry/more_fun_with_spring_scopes），其中提到PageScope的实现。

有了Scope的实现类之后，我们需要把这个Scope注册到容器中，才能供相应的bean定义使用。通常情况下，我们可以使用ConfigurableBeanFactory的以下方法注册自定义scope：

```
void registerScope(String scopeName, Scope scope);
```

其中，参数scopeName就是使用的bean定义可以指定的名称，比如Spring框架默认提供的自定义scope类型request或者session。参数scope即我们提供的Scope实现类实例。

对于以上的ThreadScope，如果容器为BeanFactory类型（当然，更应该实现ConfigurableBeanFactory），我们可以通过如下方式来注册该Scope：

```
Scope threadScope = new ThreadScope();
beanFactory.registerScope("thread",threadScope);
```

之后，我们就可以在需要的bean定义中直接通过“thread”名称来指定该bean定义对应的scope为以上注册的ThreadScope了，如以下代码所示：

```
<bean id="beanName" class="..." scope="thread"/>
```

除了直接编码调用ConfigurableBeanFactory的registerScope来注册scope，Spring还提供了一个专门用于统一注册自定义scope的BeanFactoryPostProcessor实现（有关BeanFactoryPostProcessor的更多细节稍后将详述），即org.springframework.beans.factory.config.CustomScopeConfigurer。对于ApplicationContext来说，因为它可以自动识别并加载BeanFactoryPostProcessor，所以我们就可以直接在配置文件中，通过这个CustomScopeConfigurer注册来ThreadScope（如代码清单4-29所示）。

代码清单4-29　使用CustomScopeConfigurer注册自定义scope

```
<bean class="org.springframework.beans.factory.config.CustomScopeConfigurer">
  <property name="scopes">
    <map>
      <entry key="thread" value="com.foo.ThreadScope"/>
    </map>
  </property>
</bean>
```

在以上工作全部完成之后，我们就可以在自己的bean定义中使用这个新增加到容器的自定义scope“thread”了，如下代码演示了通常情况下“thread”自定义scope的使用：

```
<bean id="beanName" class="..." scope="thread">
  <aop:scoped-proxy/>
</bean>
```

由于<aop:scoped-proxy/>涉及Spring AOP相关知识，这里不会详细讲述。需要注意的是，使用了自定义scope的bean定义，需要该元素来为其在合适的时间创建和销毁相应的代理对象实例。对于request、session和global session来说，也是如此。

4.3.6　工厂方法与 FactoryBean

在强调“面向接口编程”的同时，有一点需要注意：虽然对象可以通过声明接口来避免对特定接口实现类的过度耦合，但总归需要一种方式将声明依赖接口的对象与接口实现类关联起来。否则，只依赖一个不做任何事情的接口是没有任何用处的。假设我们有一个像代码清单4-30所声明的Foo类，它声明了一个BarInterface依赖。

代码清单4-30 依赖于某一BarInterface接口的Foo类定义

```
public class Foo
{
  private BarInterface barInstance;
  public Foo()
  {
    // 我们应该避免这样做
    // instance = new BarInterfaceImpl();
  }
  // ...
}
```

如果该类是由我们设计并开发的，那么还好说，我们可以通过依赖注入，让容器帮助我们解除接口与实现类之间的耦合性。但是，有时，我们需要依赖第三方库，需要实例化并使用第三方库中的相关类，这时，接口与实现类的耦合性需要其他方式来避免。

通常的做法是通过使用工厂方法（Factory Method）模式，提供一个工厂类来实例化具体的接口实现类，这样，主体对象只需要依赖工厂类，具体使用的实现类有变更的话，只是变更工厂类，而主体对象不需要做任何变动。代码清单4-31演示了这种做法。

代码清单4-31 使用了工厂方法模式的Foo类可能定义

```
public class Foo
{
  private BarInterface barInterface;
  public Foo()
  {
    // barInterface = BarInterfaceFactory.getInstance();
    // 或者
    // barInterface = new BarInterfaceFactory().getInstance();
  }
  ...
}
```

针对使用工厂方法模式实例化对象的方式，Spring的IoC容器同样提供了对应的集成支持。我们所要做的，只是将工厂类所返回的具体的接口实现类注入给主体对象（这里是Foo）。

注意 有关工厂方法模式的信息，可以参考设计模式方面的书籍或者网上有关资源。

1. 静态工厂方法（Static Factory Method）

假设某个第三方库发布了BarInterface，为了向使用该接口的客户端对象屏蔽以后可能对BarInterface实现类的变动，同时还提供了一个静态的工厂方法实现类StaticBarInterface-Factory，代码如下：

```
public class StaticBarInterfaceFactory
{
  public static BarInterface getInstance()
  {
```

```
        return new BarInterfaceImpl();
    }
}
```

为了将该静态工厂方法类返回的实现注入Foo，我们使用以下方式进行配置（通过setter方法注入方式为Foo注入BarInterface的实例）：

```
<bean id="foo" class="...Foo">
  <property name="barInterface">
    <ref bean="bar"/>
  </property>
</bean>

<bean id="bar" class="...StaticBarInterfaceFactory" factory-method="getInstance"/>
```

class指定静态方法工厂类，factory-method指定工厂方法名称，然后，容器调用该静态方法工厂类的指定工厂方法（getInstance），并返回方法调用后的结果，即BarInterfaceImpl的实例。也就是说，为foo注入的bar实际上是BarInterfaceImpl的实例，即方法调用后的结果，而不是静态工厂方法类（StaticBarInterfaceFactory）。我们可以实现自己的静态工厂方法类返回任意类型的对象实例，但工厂方法类的类型与工厂方法返回的类型没有必然的相同关系。

某些时候，有的工厂类的工厂方法可能需要参数来返回相应实例，而不一定非要像我们的getInstance()这样没有任何参数。对于这种情况，可以通过<constructor-arg>来指定工厂方法需要的参数，比如现在StaticBarInterfaceFactory需要其他依赖来返回某个BarInterface的实现，其定义可能如下：

```
public class StaticBarInterfaceFactory
{
  public static BarInterface getInstance(Foobar foobar)
  {
    return new BarInterfaceImpl(foobar);
  }
}
```

为了让包含方法参数的工厂方法能够预期返回相应的实现类实例，我们可以像代码清单4-32所演示的那样，通过<constructor-arg>为工厂方法传入相应参数。

代码清单4-32　使用<constructor-arg>调用含有参数的工厂方法

```
<bean id="foo" class="...Foo">
  <property name="barInterface">
    <ref bean="bar"/>
  </property>
</bean>

<bean id="bar" class="...StaticBarInterfaceFactory" factory-method="getInstance">
  <constructor-arg>
    <ref bean="foobar"/>
  </constructor-arg>
</bean>

<bean id="foobar" class="...FooBar"/>
```

唯一需要注意的就是，针对静态工厂方法实现类的bean定义，使用<constructor-arg>传入的是工厂方法的参数，而不是静态工厂方法实现类的构造方法的参数。（况且，静态工厂方法实现类也没有提供显式的构造方法。）

2. 非静态工厂方法（Instance Factory Method）

既然可以将静态工厂方法实现类的工厂方法调用结果作为bean注册到容器中，我们同样可以针对基于工厂类实例的工厂方法调用结果应用相同的功能，只不过，表达方式可能需要稍微变一下。

现在为BarInterface提供非静态的工厂方法实现类，该类定义如下代码所示：

```
public class NonStaticBarInterfaceFactory
{
  public BarInterface getInstance()
  {
    return new BarInterfaceImpl();
  }
  ...
}
```

因为工厂方法为非静态的，我们只能通过某个NonStaticBarInterfaceFactory实例来调用该方法（哦，错了，是容器来调用），那么也就有了如下的配置内容：

```
<bean id="foo" class="...Foo">
  <property name="barInterface">
    <ref bean="bar"/>
  </property>
</bean>

<bean id="barFactory" class="...NonStaticBarInterfaceFactory"/>

<bean id="bar" factory-bean="barFactory" factory-method="getInstance"/>
```

NonStaticBarInterfaceFactory是作为正常的bean注册到容器的，而bar的定义则与静态工厂方法的定义有些不同。现在使用factory-bean属性来指定工厂方法所在的工厂类实例，而不是通过class属性来指定工厂方法所在类的类型。指定工厂方法名则相同，都是通过factory-method属性进行的。

如果非静态工厂方法调用时也需要提供参数的话，处理方式是与静态的工厂方法相似的，都可以通过<constructor-arg>来指定方法调用参数。

3. FactoryBean

FactoryBean是Spring容器提供的一种可以扩展容器对象实例化逻辑的接口，请不要将其与容器名称BeanFactory相混淆。FactoryBean，其主语是Bean，定语为Factory，也就是说，它本身与其他注册到容器的对象一样，只是一个Bean而已，只不过，这种类型的Bean本身就是生产对象的工厂（Factory）。

当某些对象的实例化过程过于烦琐，通过XML配置过于复杂，使我们宁愿使用Java代码来完成这个实例化过程的时候，或者，某些第三方库不能直接注册到Spring容器的时候，就可以实现org.springframework.beans.factory.FactoryBean接口，给出自己的对象实例化逻辑代码。当然，不使用FactoryBean，而像通常那样实现自定义的工厂方法类也是可以的。不过，FactoryBean可是Spring提供的对付这种情况的“制式装备”[①]哦！

要实现并使用自己的FactoryBean其实很简单，org.springframework.beans.factory.FactoryBean只定义了三个方法，如以下代码所示：

```
public interface FactoryBean {
```

① 制式装备通常指正规军使用的标准装备。

```
    Object getObject() throws Exception;
    Class getObjectType();
    boolean isSingleton();
}
```

getObject()方法会返回该FactoryBean"生产"的对象实例，我们需要实现该方法以给出自己的对象实例化逻辑；getObjectType()方法仅返回getObject()方法所返回的对象的类型，如果预先无法确定，则返回null；isSingleton()方法返回结果用于表明，工厂方法（getObject()）所"生产"的对象是否要以singleton形式存在于容器中。如果以singleton形式存在，则返回true，否则返回false；

如果我们想每次得到的日期都是第二天，可以实现一个如代码清单4-33所示的FactoryBean。

代码清单4-33　NextDayDateFactoryBean的定义代码

```
import org.joda.time.DateTime;
import org.springframework.beans.factory.FactoryBean;

public class NextDayDateFactoryBean implements FactoryBean {

    public Object getObject() throws Exception {
        return new DateTime().plusDays(1);
    }

    public Class getObjectType() {
        return DateTime.class;
    }

    public boolean isSingleton() {
        return false;
    }

}
```

很简单的实现，不是嘛？

要使用NextDayDateFactoryBean，只需要如下这样将其注册到容器即可：

```
<bean id="nextDayDateDisplayer" class="...NextDayDateDisplayer">
    <property name="dateOfNextDay">
        <ref bean="nextDayDate"/>
    </property>
</bean>

<bean id="nextDayDate" class="...NextDayDateFactoryBean">
</bean>
```

配置上看不出与平常的bean定义有何不同，不过，只有当我们看到NextDayDateDisplayer的定义的时候，才会知道FactoryBean的魔力到底在哪。NextDayDateDisplayer的定义如下：

```
public class NextDayDateDisplayer
{
    private DateTime dateOfNextDay;
    // 相应的setter方法
    // ...
}
```

看到了嘛？NextDayDateDisplayer所声明的依赖dateOfNextDay的类型为DateTime，而不是NextDayDateFactoryBean。也就是说FactoryBean类型的bean定义，通过正常的id引用，容器返回

的是FactoryBean所“生产”的对象类型，而非FactoryBean实现本身。

如果一定要取得FactoryBean本身的话，可以通过在bean定义的id之前加前缀&来达到目的。代码清单4-34展示了获取FactoryBean本身与获取FactoryBean“生产”的对象之间的差别。

代码清单4-34　使用&获取FactoryBean的实例演示

```
Object nextDayDate = container.getBean("nextDayDate");
assertTrue(nextDayDate instanceof DateTime);

Object factoryBean = container.getBean("&nextDayDate");
assertTrue(factoryBean instanceof FactoryBean);
assertTrue(factoryBean instanceof NextDayDateFactoryBean);

Object factoryValue = ((FactoryBean)factoryBean).getObject();
assertTrue(factoryValue instanceof DateTime);

assertNotSame(nextDayDate, factoryValue);
assertEquals(((DateTime)nextDayDate).getDayOfYear(),((DateTime)factoryValue).getDayOfYear());
```

Spring容器内部许多地方了使用FactoryBean。下面是一些比较常见的FactoryBean实现，你可以参照FactoryBean的Javadoc以了解更多内容。

- JndiObjectFactoryBean
- LocalSessionFactoryBean
- SqlMapClientFactoryBean
- ProxyFactoryBean
- TransactionProxyFactoryBean

4.3.7　偷梁换柱之术

在学习以下内容之前，先提一下有关bean的scope的使用“陷阱”，特别是prototype在容器中的使用，以此引出本节将要介绍的Spring容器较为独特的功能特性：方法注入（Method Injection）以及方法替换（Method Replacement）。

我们知道，拥有prototype类型scope的bean，在请求方每次向容器请求该类型对象的时候，容器都会返回一个全新的该对象实例。为了简化问题的叙述，我们直接将FX News系统中的FXNewsBean定义注册到容器中，并将其scope设置为prototype。因为它是有状态的类型，每条新闻都应该是新的独立个体；同时，我们给出MockNewsPersister类，使其实现IFXNewsPersister接口，以模拟注入FXNewsBean实例后的情况。这样，我们就有了代码清单4-35所展示的类声明和相关配置。

代码清单4-35　MockNewsPersister的定义以及相关配置

```
public class MockNewsPersister implements IFXNewsPersister {
  private FXNewsBean newsBean;

  public void persistNews(FXNewsBean bean) {
    persistNewes();
  }
  public void persistNews()
  {
    System.out.println("persist bean:"+getNewsBean());
  }
  public FXNewsBean getNewsBean() {
    return newsBean;
  }
```

```
    public void setNewsBean(FXNewsBean newsBean) {
        this.newsBean = newsBean;
    }
}
配置为
<bean id="newsBean" class="..domain.FXNewsBean" singleton="false">
</bean>
<bean id="mockPersister" class="..impl.MockNewsPersister">
    <property name="newsBean">
        <ref bean="newsBean"/>
    </property>
</bean>
```

当多次调用MockNewsPersister的persistNews时，你猜会得到什么结果？如下代码可以帮助我们揭开答案：

```
BeanFactory container = new XmlBeanFactory(new ClassPathResource(".."));
MockNewsPersister persister = (MockNewsPersister)container.getBean("mockPersister");
persister.persistNews();
persister.persistNews();
输出:
persist bean:..domain.FXNewsBean@1662dc8
persist bean:..domain.FXNewsBean@1662dc8
```

从输出看，对象实例是相同的，而这与我们的初衷是相悖的。因为每次调用persistNews都会调用getNewsBean()方法并返回一个FXNewsBean实例，而FXNewsBean实例是prototype类型的，因此每次不是应该输出不同的对象实例嘛？

好了，问题实际上不是出在FXNewsBean的scope类型是否是prototype的，而是出在实例的取得方式上面。虽然FXNewsBean拥有prototype类型的scope，但当容器将一个FXNewsBean的实例注入MockNewsPersister之后，MockNewsPersister就会一直持有这个FXNewsBean实例的引用。虽然每次输出都调用了getNewsBean()方法并返回了 FXNewsBean 的实例，但实际上每次返回的都是MockNewsPersister持有的容器第一次注入的实例。这就是问题之所在。换句话说，第一个实例注入后，MockNewsPersister再也没有重新向容器申请新的实例。所以，容器也不会重新为其注入新的FXNewsBean类型的实例。

知道原因之后，我们就可以解决这个问题了。解决问题的关键在于保证getNewsBean()方法每次从容器中取得新的FXNewsBean实例，而不是每次都返回其持有的单一实例。

1. 方法注入

Spring容器提出了一种叫做方法注入（Method Injection）的方式，可以帮助我们解决上述问题。我们所要做的很简单，只要让getNewsBean方法声明符合规定的格式，并在配置文件中通知容器，当该方法被调用的时候，每次返回指定类型的对象实例即可。方法声明需要符合的规格定义如下：

```
<public|protected> [abstract] <return-type> theMethodName(no-arguments);
```

也就是说，该方法必须能够被子类实现或者覆写，因为容器会为我们要进行方法注入的对象使用Cglib动态生成一个子类实现，从而替代当前对象。既然我们的getNewsBean()方法已经满足以上方法声明格式，剩下唯一要做的就是配置该类，配置内容如下所示：

```
<bean id="newsBean" class="..domain.FXNewsBean" singleton="false">
</bean>
<bean id="mockPersister" class="..impl.MockNewsPersister">
    <lookup-method name="getNewsBean" bean="newsBean"/>
</bean>
```

通过<lookup-method>的name属性指定需要注入的方法名，bean属性指定需要注入的对象，当getNewsBean方法被调用的时候，容器可以每次返回一个新的FXNewsBean类型的实例。所以，这个时候，我们再次检查执行结果，输出的实例引用应该是不同的：

```
persist bean:..domain.FXNewsBean@18aaa1e
persist bean:..domain.FXNewsBean@a6aeed
```

哇噢，很帅不是吗？

注意　FXNewsBean的取得实际上可以在相应方法中按需要自行实例化，而不一定非要注册到容器中，从容器中获取。我们只是为了引入prototype的使用“陷阱”以及方法注入功能，才将FXNewsBean以prototype类型注册到容器中供使用。当然，如果愿意你也可以以这种方式使用。在最终输出的结果中，对象引用的数字不一定就是上面的那样。因为每次注入的实例是不同的，所以对应实例的数字也可能不同。在此只需要关注每次同时输出的结果是否相同即可说明问题。

2. 殊途同归

除了使用方法注入来达到“每次调用都让容器返回新的对象实例”的目的，还可以使用其他方式达到相同的目的。下面给出其他两种解决类似问题的方法，供读者参考。

- 使用BeanFactoryAware接口

我们知道，即使没有方法注入，只要在实现getNewsBean()方法的时候，能够保证每次调用BeanFactory的getBean("newsBean")，就同样可以每次都取得新的FXNewsBean对象实例。现在，我们唯一需要的，就是让MockNewsPersister拥有一个BeanFactory的引用。

Spring框架提供了一个BeanFactoryAware接口，容器在实例化实现了该接口的bean定义的过程中，会自动将容器本身注入该bean。这样，该bean就持有了它所处的BeanFactory的引用。BeanFactoryAware的定义如下代码所示：

```
public interface BeanFactoryAware {
  void setBeanFactory(BeanFactory beanFactory) throws BeansException;
}
```

我们让MockNewsPersister实现该接口以持有其所处的BeanFactory的引用，这样MockNewsPersister的定义如代码清单4-36所示。

代码清单4-36　实现了BeanFactoryAware接口的MockNewsPersister及相关配置

```
public class MockNewsPersister implements IFXNewsPersister,BeanFactoryAware {
  private BeanFactory beanFactory;

  public void setBeanFactory(BeanFactory bf) throws BeansException {
    this.beanFactory = bf;
  }
  public void persistNews(FXNewsBean bean) {
    persistNews();
  }
  public void persistNews()
  {
    System.out.println("persist bean:"+getNewsBean());
  }
  public FXNewsBean getNewsBean() {
    return beanFactory.getBean("newsBean");
```

```
    }
}
配置简化为
<bean id="newsBean" class="..domain.FXNewsBean" singleton="false">
</bean>
<bean id="mockPersister" class="..impl.MockNewsPersister">
</bean>
```

如此，可以预见到，输出的结果将与我们所预期的相同：

```
persist bean:..domain.FXNewsBean@121cc40
persist bean:..domain.FXNewsBean@1e893df
```

实际上，方法注入动态生成的子类，完成的是与以上类似的逻辑，只不过实现细节上不同而已。

- 使用ObjectFactoryCreatingFactoryBean

ObjectFactoryCreatingFactoryBean是Spring提供的一个FactoryBean实现，它返回一个ObjectFactory实例。从ObjectFactoryCreatingFactoryBean返回的这个ObjectFactory实例可以为我们返回容器管理的相关对象。实际上，ObjectFactoryCreatingFactoryBean实现了BeanFactoryAware接口，它返回的ObjectFactory实例只是特定于与Spring容器进行交互的一个实现而已。使用它的好处就是，隔离了客户端对象对BeanFactory的直接引用。

现在，我们使用ObjectFactory取得FXNewsBean的实例，代码清单4-37给出了对应这种方式的MockNewsPersister实现声明。

代码清单4-37　使用ObjectFactory的MockNewsPersister定义

```
public class MockNewsPersister implements IFXNewsPersister {
    private ObjectFactory newsBeanFactory;

    public void persistNews(FXNewsBean bean) {
        persistNews();
    }
    public void persistNews()
    {
        System.out.println("persist bean:"+getNewsBean());
    }
    public FXNewsBean getNewsBean() {
        return newsBeanFactory.getObject();
    }
    public void setNewsBeanFactory(ObjectFactory newsBeanFactory) {
        this.newsBeanFactory = newsBeanFactory;
    }
}
```

有了以上的类定义之后，我们应该为MockNewsPersister注入相应的ObjectFactory，这也正是ObjectFactoryCreatingFactoryBean闪亮登场的时候，代码清单4-38给出了对应的配置内容。

代码清单4-38　使用ObjectFactoryCreatingFactoryBean的相关配置

```
<bean id="newsBean" class="..domain.FXNewsBean" singleton="false">
</bean>
<bean id="newsBeanFactory" class="org.springframework.beans.factory.config. ➥
ObjectFactoryCreatingFactoryBean">
    <property name="targetBeanName">
        <idref bean="newsBean"/>
    </property>
</bean>
<bean id="mockPersister" class="..impl.MockNewsPersister">
```

```
    <property name="newsBeanFactory">
      <ref bean="newsBeanFactory"/>
    </property>
</bean>
```

看，真有效！

```
persist bean:..domain.FXNewsBean@ecd7e
persist bean:..domain.FXNewsBean@1d520c4
```

> **提示** 也可以使用ServiceLocatorFactoryBean来代替ObjectFactoryCreatingFactory-Bean，该FactoryBean可以让我们自定义工厂接口，而不用非要使用Spring的ObjectFactory。可以参照该类定义的Javadoc取得更多信息，Javadoc中有详细的实例，足够让你了解该类的使用和功能。

3. 方法替换

与方法注入只是通过相应方法为主体对象注入依赖对象不同，方法替换更多体现在方法的实现层面上，它可以灵活替换或者说以新的方法实现覆盖掉原来某个方法的实现逻辑。基本上可以认为，方法替换可以帮助我们实现简单的方法拦截功能。要知道，我们现在可是在不知不觉中迈上了AOP的大道哦！

假设某天我看FXNewsProvider不爽，想替换掉它的getAndPersistNews方法默认逻辑，这时，我就可以用方法替换将它的原有逻辑给替换掉。

> **小心** 这里只是为了演示方法替换（Method Replacement）的功能，不要真的这么做。要使用也要用在好的地方，对吧？

首先，我们需要给出org.springframework.beans.factory.support.MethodReplacer的实现类，在这个类中实现将要替换的方法逻辑。假设我们只是简单记录日志，打印简单信息，那么就可以给出一个类似代码清单4-39所示的MethodReplacer实现类。

代码清单4-39 FXNewsProviderMethodReplacer类的定义

```
public class FXNewsProviderMethodReplacer implements MethodReplacer {

  private static final transient Log logger = ➥
  LogFactory.getLog(FXNewsProviderMethodReplacer.class);

  public Object reimplement(Object target, Method method, Object[] args) ➥
  throws Throwable {
    logger.info("before executing method["+method.getName()+➥
    "] on Object["+target.getClass().getName()+"].");

    System.out.println("sorry,We will do nothing this time.");

    logger.info("end of executing method["+method.getName()+➥
    "] on Object["+target.getClass().getName()+"].");
    return null;
  }
}
```

有了要替换的逻辑之后，我们就可以把这个逻辑通过<replaced-method>配置到FXNewsProvider的bean定义中，使其生效，配置内容如代码清单4-40所示。

代码清单4-40　FXNewsProvider中使用方法替换的相关配置

```
<bean id="djNewsProvider" class="..FXNewsProvider">
  <constructor-arg index="0">
    <ref bean="djNewsListener"/>
  </constructor-arg>
  <constructor-arg index="1">
    <ref bean="djNewsPersister"/>
  </constructor-arg>
  <replaced-method name="getAndPersistNews" replacer="providerReplacer">
  </replaced-method>
</bean>

<bean id="providerReplacer" class="..FXNewsProviderMethodReplacer">
</bean>

<!--其他bean配置-->
...
```

现在，你猜调用FXNewsProvider的getAndPersistNews方法后，会得到什么结果？输出结果如下所示：

```
771 [main] INFO ..FXNewsProviderMethodReplacer
  - before executing method[getAndPersistNews]
    on Object[..FXNewsProvider$$EnhancerByCGLIB$$3fa709d3].
sorry,We will do nothing this time.
771 [main] INFO ..FXNewsProviderMethodReplacer
  - end of executing method[getAndPersistNews]
    on Object[..FXNewsProvider$$EnhancerByCGLIB$$3fa709d3].
```

我们把FXNewsProvider的getAndPersistNews方法逻辑给完全替换掉了。现在该方法基本上什么也没做，哇……

最后需要强调的是，这种方式刚引入的时候执行效率不是很高。而且，当你充分了解并应用Spring AOP之后，我想你也不会再回头求助这个特色功能。不过，怎么说这也是一个选择，场景合适的话，为何不用呢？

哦，如果要替换的方法存在参数，或者对象存在多个重载的方法，可以在<replaced-method>内部通过<arg-type>明确指定将要替换的方法参数类型。祝“替换”愉快！

4.4　容器背后的秘密

子曰：学而不思则罔。除了了解Spring的IoC容器如何使用，了解Spring的IoC容器都提供了哪些功能，我们也应该想一下，Spring的IoC容器内部到底是如何来实现这些的呢？虽然我们不太可能“重新发明轮子”，但是，如图4-7（该图摘自Spring官方参考文档）所示的那样，只告诉你“Magic Happens Here”，你是否就能心满意足呢？

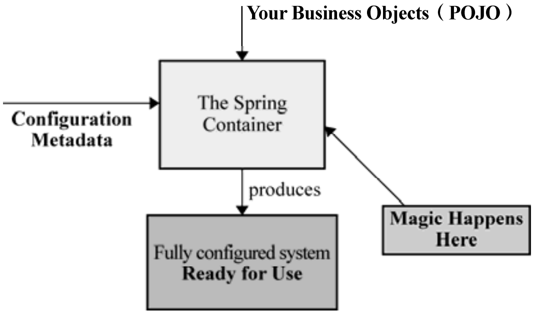

图4-7 即将揭示的奥秘所在

好，如果你的答案是“不”（我当然认为你说的是“不想一直被蒙在鼓里”），那么就随我一起来探索一下这个“黑匣子”里面到底有些什么……

4.4.1 “战略性观望”

Spring的IoC容器所起的作用，就像图4-7所展示的那样，它会以某种方式加载Configuration Metadata（通常也就是XML格式的配置信息），然后根据这些信息绑定整个系统的对象，最终组装成一个可用的基于轻量级容器的应用系统。

Spring的IoC容器实现以上功能的过程，基本上可以按照类似的流程划分为两个阶段，即容器启动阶段和Bean实例化阶段，如图4-8所示。

Spring的IoC容器在实现的时候，充分运用了这两个实现阶段的不同特点，在每个阶段都加入了相应的容器扩展点，以便我们可以根据具体场景的需要加入自定义的扩展逻辑。

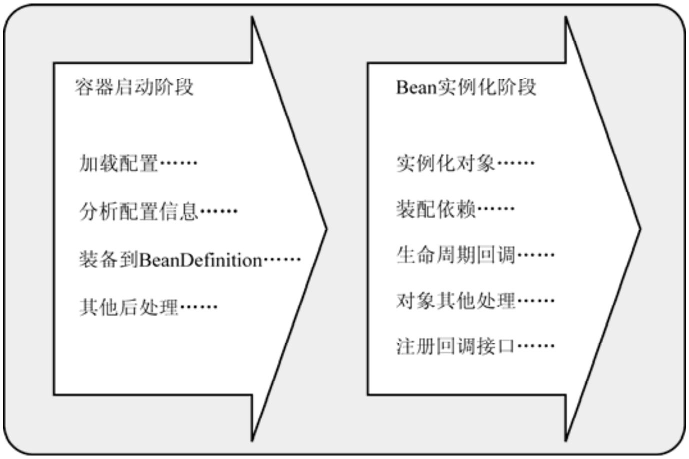

图4-8 容器功能实现的各个阶段

1. 容器启动阶段

容器启动伊始，首先会通过某种途径加载Configuration MetaData。除了代码方式比较直接，在大部分情况下，容器需要依赖某些工具类（`BeanDefinitionReader`）对加载的Configuration MetaData

进行解析和分析，并将分析后的信息编组为相应的BeanDefinition，最后把这些保存了bean定义必要信息的BeanDefinition，注册到相应的BeanDefinitionRegistry，这样容器启动工作就完成了。图4-9演示了这个阶段的主要工作。

图4-9　XML配置信息到BeanDefinition的映射

总地来说，该阶段所做的工作可以认为是准备性的，重点更加侧重于对象管理信息的收集。当然，一些验证性或者辅助性的工作也可以在这个阶段完成。

2. Bean实例化阶段

经过第一阶段，现在所有的bean定义信息都通过BeanDefinition的方式注册到了BeanDefinitionRegistry中。当某个请求方通过容器的getBean方法明确地请求某个对象，或者因依赖关系容器需要隐式地调用getBean方法时，就会触发第二阶段的活动。

该阶段，容器会首先检查所请求的对象之前是否已经初始化。如果没有，则会根据注册的BeanDefinition所提供的信息实例化被请求对象，并为其注入依赖。如果该对象实现了某些回调接口，也会根据回调接口的要求来装配它。当该对象装配完毕之后，容器会立即将其返回请求方使用。如果说第一阶段只是根据图纸装配生产线的话，那么第二阶段就是使用装配好的生产线来生产具体的产品了。

4.4.2　插手“容器的启动”

Spring提供了一种叫做BeanFactoryPostProcessor的容器扩展机制。该机制允许我们在容器实例化相应对象之前，对注册到容器的BeanDefinition所保存的信息做相应的修改。这就相当于在容器实现的第一阶段最后加入一道工序，让我们对最终的BeanDefinition做一些额外的操作，比如修改其中bean定义的某些属性，为bean定义增加其他信息等。

如果要自定义实现BeanFactoryPostProcessor，通常我们需要实现org.springframework.beans.factory.config.BeanFactoryPostProcessor接口。同时，因为一个容器可能拥有多个BeanFactoryPostProcessor，这个时候可能需要实现类同时实现Spring的org.springframework.core.Ordered接口，以保证各个BeanFactoryPostProcessor可以按照预先设定的顺序执行（如果顺序紧要的话）。但是，因为Spring已经提供了几个现成的BeanFactoryPostProcessor实现类，所以，大多时候，我们很少自己去实现某个BeanFactoryPostProcessor。其中，org.springframework.beans.factory.config.PropertyPlaceholderConfigurer和org.springframework.beans.factory.config.Property OverrideConfigurer是两个比较常用的BeanFactoryPostProcessor。另外，为了处理配置文件中的数据类型与真正的业务对象所定义的数据类型转换，Spring还允许我们通过org.springframework.beans.factory.config.CustomEditorConfigurer来注册自定义的PropertyEditor以补助容器中默认的PropertyEditor。可以参考BeanFactoryPostProcessor的Javadoc来了解更多其实现子类的情况。

我们可以通过两种方式来应用BeanFactoryPostProcessor，分别针对基本的IoC容器BeanFactory和较为先进的容器ApplicationContext。

对于BeanFactory来说，我们需要用手动方式应用所有的BeanFactoryPostProcessor，代码清单4-41演示了具体的做法。

代码清单4-41 手动装配BeanFactory使用的BeanFactoryPostProcessor

```
// 声明将被后处理的BeanFactory实例
ConfigurableListableBeanFactory beanFactory = new XmlBeanFactory(new ClassPathResource("..."));
// 声明要使用的BeanFactoryPostProcessor
PropertyPlaceholderConfigurer propertyPostProcessor = new PropertyPlaceholderConfigurer();
propertyPostProcessor.setLocation(new ClassPathResource("..."));
// 执行后处理操作
propertyPostProcessor.postProcessBeanFactory(beanFactory);
```

如果拥有多个BeanFactoryPostProcessor，我们可以添加更多类似的代码来应用所有的这些BeanFactoryPostProcessor。

对于ApplicationContext来说，情况看起来要好得多。因为ApplicationContext会自动识别配置文件中的BeanFactoryPostProcessor并应用它，所以，相对于BeanFactory，在ApplicationContext中加载并应用BeanFactoryPostProcessor，仅需要在XML配置文件中将这些BeanFactoryPost-Processor简单配置一下即可。只要如代码清单4-42所示，将相应BeanFactoryPostProcessor实现类添加到配置文件，ApplicationContext将自动识别并应用它。

代码清单4-42 通过ApplicationContext使用BeanFactoryPostProcessor

```
...
<beans>
   <bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
      <property name="locations">
         <list>
            <value>conf/jdbc.properties</value>
            <value>conf/mail.properties</value>
         </list>
      </property>
   </bean>
   ...
</beans>
```

下面让我们看一下Spring提供的这几个BeanFactoryPostProcessor实现都可以完成什么功能。

1. **PropertyPlaceholderConfigurer**

通常情况下，我们不想将类似于系统管理相关的信息同业务对象相关的配置信息混杂到XML配置文件中，以免部署或者维护期间因为改动繁杂的XML配置文件而出现问题。我们会将一些数据库连接信息、邮件服务器等相关信息单独配置到一个properties文件中，这样，如果因系统资源变动的话，只需要关注这些简单properties配置文件即可。

PropertyPlaceholderConfigurer允许我们在XML配置文件中使用占位符（PlaceHolder），并将这些占位符所代表的资源单独配置到简单的properties文件中来加载。以数据源的配置为例，使用了PropertyPlaceholderConfigurer之后（这里沿用代码清单4-42的配置内容），可以在XML配置文件中按照代码清单4-43所示的方式配置数据源，而不用将连接地址、用户名和密码等都配置到XML中。

代码清单4-43 使用了占位符的数据源配置

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" ➥
destroy-method="close">
```

```
    <property name="url">
      <value>${jdbc.url}</value>
    </property>
    <property name="driverClassName">
      <value>${jdbc.driver}</value>
    </property>
    <property name="username">
      <value>${jdbc.username}</value>
    </property>
    <property name="password">
      <value>${jdbc.password}</value>
    </property>
    <property name="testOnBorrow">
      <value>true</value>
    </property>
    <property name="testOnReturn">
      <value>true</value>
    </property>
    <property name="testWhileIdle">
      <value>true</value>
    </property>
    <property name="minEvictableIdleTimeMillis">
      <value>180000</value>
    </property>
    <property name="timeBetweenEvictionRunsMillis">
      <value>360000</value>
    </property>
    <property name="validationQuery">
      <value>SELECT 1</value>
    </property>
    <property name="maxActive">
      <value>100</value>
    </property>
</bean>
```

如果你使用过Ant或者Velocity等工具，就会发现${property}之类的表达很熟悉。现在，所有这些占位符所代表的资源，都放到了jdbc.properties文件中，如下所示：

```
jdbc.url=jdbc:mysql://server/MAIN?useUnicode=true&characterEncoding=ms932&     ➥
failOverReadOnly=false&roundRobinLoadBalance=true
jdbc.driver=com.mysql.jdbc.Driver
jdbc.username=your username
jdbc.password=your password
```

基本机制就是之前所说的那样。当BeanFactory在第一阶段加载完成所有配置信息时，BeanFactory中保存的对象的属性信息还只是以占位符的形式存在，如${jdbc.url}、${jdbc.driver}。当PropertyPlaceholderConfigurer作为BeanFactoryPostProcessor被应用时，它会使用properties配置文件中的配置信息来替换相应BeanDefinition中占位符所表示的属性值。这样，当进入容器实现的第二阶段实例化bean时，bean定义中的属性值就是最终替换完成的了。

PropertyPlaceholderConfigurer不单会从其配置的properties文件中加载配置项，同时还会检查Java的System类中的Properties，可以通过setSystemPropertiesMode()或者setSystemPropertiesModeName()来控制是否加载或者覆盖System相应Properties的行为。PropertyPlaceholderConfigurer提供了SYSTEM_PROPERTIES_MODE_FALLBACK、SYSTEM_PROPERTIES_MODE_NEVER和SYSTEM_PROPERTIES_MODE_OVERRIDE三种模式。默认采用的是SYSTEM_PROPERTIES_ MODE_FALLBACK，即如

果properties文件中找不到相应配置项，则到System的Properties中查找，我们还可以选择不检查System的Properties或者覆盖它。更多信息请参照PropertyPlaceholderConfigurer的Javadoc文档。

2. **PropertyOverrideConfigurer**

PropertyPlaceholderConfigurer可以通过占位符，来明确表明bean定义中的property与properties文件中的各配置项之间的对应关系。如果说PropertyPlaceholderConfigurer做的这些是“明事”的话，那相对来说，PropertyOverrideConfigurer所做的可能就有点儿“神不知鬼不觉”了。

可以通过PropertyOverrideConfigurer对容器中配置的任何你想处理的bean定义的property信息进行覆盖替换。这听起来比较抽象，我们还是给个例子吧！比如之前的dataSource定义中，maxActive的值为100，如果我们觉得100不合适，那么可以通过PropertyOverrideConfigurer在其相应的properties文件中做如下所示配置，把100这个值给覆盖掉，如将其配置为200：

```
dataSource.maxActive=200
```

这样，当容器实例化对象的时候，该dataSource对象对应的maxActive值就是200，而不是原来XML配置中的100。也就是说，PropertyOverrideConfigurer的properties文件中的配置项，覆盖掉了原来XML中的bean定义的property信息。但这样的活动，只看XML配置的话，你根本看不出哪个bean定义的哪个property会被覆盖替换掉，只有查看PropertyOverrideConfigurer指定的properties配置文件才会了解。基本上，这种覆盖替换对于bean定义来说是透明的。

如果要对容器中的某些bean定义的property信息进行覆盖，我们需要按照如下规则提供一个PropertyOverrideConfigurer使用的配置文件：

```
beanName.propertyName=value
```

也就是说，properties文件中的键是以XML中配置的bean定义的beanName为标志开始的（通常就是id指定的值），后面跟着相应被覆盖的property的名称，比如上面的maxActive。

下面是针对dataSource定义给出的PropertyOverrideConfigurer的propeties文件配置信息：

```
# pool-adjustment.properties
dataSource.minEvictableIdleTimeMillis=1000
dataSource.maxActive=50
```

这样，当按照如下代码，将PropertyOverrideConfigurer加载到容器之后，dataSource原来定义的默认值就会被pool-adjustment.properties文件中的信息所覆盖：

```
<bean class="org.springframework.beans.factory.config.PropertyOverrideConfigurer">
  <property name="location" value="pool-adjustment.properties"/>
</bean>
```

pool-adjustment.properties中没有提供的配置项将继续使用原来XML配置中的默认值。

当容器中配置的多个PropertyOverrideConfigurer对同一个bean定义的同一个property值进行处理的时候，最后一个将会生效。

配置在properties文件中的信息通常都以明文表示，PropertyOverrideConfigurer的父类PropertyResourceConfigurer提供了一个protected类型的方法convertPropertyValue，允许子类覆盖这个方法对相应的配置项进行转换，如对加密后的字符串解密之后再覆盖到相应的bean定义中。当然，既然PropertyPlaceholderConfigurer也同样继承了PropertyResourceConfigurer，我们也可以针对PropertyPlaceholderConfigurer应用类似的功能。

3. CustomEditorConfigurer

其他两个BeanFactoryPostProcessor都是通过对BeanDefinition中的数据进行变更以达到某种目的。与它们有所不同，CustomEditorConfigurer是另一种类型的BeanFactoryPostProcessor实现，它只是辅助性地将后期会用到的信息注册到容器，对BeanDefinition没有做任何变动。

我们知道，不管对象是什么类型，也不管这些对象所声明的依赖对象是什么类型，通常都是通过XML（或者properties甚至其他媒介）文件格式来配置这些对象类型。但XML所记载的，都是String类型，即容器从XML格式的文件中读取的都是字符串形式，最终应用程序却是由各种类型的对象所构成。要想完成这种由字符串到具体对象的转换（不管这个转换工作最终由谁来做），都需要这种转换规则相关的信息，而CustomEditorConfigurer就是帮助我们传达类似信息的。

Spring内部通过JavaBean的PropertyEditor来帮助进行String类型到其他类型的转换工作。只要为每种对象类型提供一个PropertyEditor，就可以根据该对象类型取得与其相对应的PropertyEditor来做具体的类型转换。Spring容器内部在做具体的类型转换的时候，会采用JavaBean框架内默认的PropertyEditor搜寻逻辑，从而继承了对原生类型以及java.lang.String.java.awt.Color和java.awt.Font等类型的转换支持。同时，Spring框架还提供了自身实现的一些PropertyEditor，这些PropertyEditor大部分都位于org.springframework. beans.propertyeditors包下。以下是这些Spring提供的部分PropertyEditor的简要说明。

- StringArrayPropertyEditor。该PropertyEditor会将符合CSV格式的字符串转换成String[]数组的形式，默认是以逗号（，）分隔的字符串，但可以指定自定义的字符串分隔符。ByteArrayPropertyEditor、CharArrayPropertyEditor等都属于类似功能的PropertyEditor，参照Javadoc可以取得相应的详细信息。
- ClassEditor。根据String类型的class名称，直接将其转换成相应的Class对象，相当于通过Class.forName(String)完成的功效。可以通过String[]数组的形式传入需转换的值，以达到与提供的ClassArrayEditor同样的目的。
- FileEditor。Spring提供的对应java.io.File类型的PropertyEditor。同属于对资源进行定位的PropertyEditor还有InputStreamEditor、URLEditor等。
- LocaleEditor。针对java.util.Locale类型的PropertyEditor，格式可以参照LocaleEditor和Locale的Javadoc说明。
- PatternEditor。针对Java SE 1.4之后才引入的java.util.regex.Pattern的PropertyEditor，格式可以参照java.util.regex.Pattern类的Javadoc。

以上这些PropertyEditor，容器通常会默认加载使用，所以，即使我们不告诉容器应该如何对这些类型进行转换，容器同样可以正确地完成工作。但当我们需要指定的类型没有包含在以上所提到的PropertyEditor之列的时候，就需要给出针对这种类型的PropertyEditor实现，并通过CustomEditorConfigurer告知容器，以便容器在适当的时机使用到适当的PropertyEditor。

- 自定义PropertyEditor

通常情况下，对于Date类型，不同的Locale、不同的系统在表现形式上存在不同的需求。如系统这个部分需要以yyyy-MM-dd的形式表现日期，系统那个部分可能又需要以yyyyMMdd的形式对日期进行转换。虽然可以使用Spring提供的CustomDateEditor，不过为了能够演示自定义PropertyEditor的详细流程，在此我们有必要"重新发明轮子"！

下面是对自定义PropertyEditor实现的简单介绍。

给出针对特定对象类型的PropertyEditor实现

假设需要对yyyy/MM/dd形式的日期格式转换提供支持。虽然可以直接让PropertyEditor实现类去实现java.beans.PropertyEditor接口，不过，通常情况下，我们可以直接继承java.beans.PropertyEditorSupport类以避免实现java.beans.PropertyEditor接口的所有方法。就好像这次，我们仅仅让DatePropertyEditor完成从String到java.util.Date的转换，只需要实现setAsText(String)方法，而其他方法一概不管。该自定义PropertyEditor类定义如代码清单4-44所示。

代码清单4-44　DatePropertyEditor定义

```
public class DatePropertyEditor extends PropertyEditorSupport {
  private String datePattern;

  @Override
  public void setAsText(String text) throws IllegalArgumentException {
    DateTimeFormatter dateTimeFormatter = DateTimeFormat.forPattern(getDatePattern());
    Date dateValue = dateTimeFormatter.parseDateTime(text).toDate();
    setValue(dateValue);
  }
  public String getDatePattern() {
    return datePattern;
  }
  public void setDatePattern(String datePattern) {
    this.datePattern = datePattern;
  }
}
```

如果仅仅是支持单向的从String到相应对象类型的转换，只要覆写方法setAsText(String)即可。如果想支持双向转换，需要同时考虑getAsText()方法的覆写。

通过CustomEditorConfigurer注册自定义的PropertyEditor

如果有类似于DateFoo这样的类对java.util.Date类型的依赖声明，通常情况下，会以代码清单4-45所示的形式声明并将该类配置到容器中。

代码清单4-45　DateFoo的定义声明以及相关配置

```
类声明类似于
public class DateFoo {
  private Date date;

  public Date getDate() {
    return date;
  }
  public void setDate(Date date) {
    this.date = date;
  }
}
配置类似于
<bean id="dateFoo" class="...DateFoo">
  <property name="date">
    <value>2007/10/16</value>
  </property>
</bean>
```

但是，默认情况下，Spring容器找不到合适的PropertyEditor将字符串“2007/10/16”转换成对

象所声明的java.util.Date类型。所以，我们通过CustomEditorConfigurer将刚实现的DatePropertyEditor注册到容器，以告知容器按照DatePropertyEditor的形式进行String到java.util.Date类型的转换工作。

如果使用的容器是BeanFactory的实现，比如XmlBeanFactory，就需要通过编码手动应用CustomEditorConfigurer到容器，类似如下形式：

```
XmlBeanFactory beanFactory = new XmlBeanFactory(new ClassPathResource("..."));
//
CustomEditorConfigurer ceConfigurer = new CustomEditorConfigurer();
Map customerEditors = new HashMap();
customerEditors.put(java.util.Date.class, new DatePropertyEditor());
ceConfigurer.setCustomEditors(customerEditors);
//
ceConfigurer.postProcessBeanFactory(beanFactory);
```

但如果使用的是ApplicationContext相应实现，因为ApplicationContext会自动识别BeanFactoryPostProcessor并应用，所以只需要在相应配置文件中配置一下，如代码清单4-46所示。

代码清单4-46　使用CustomEditorConfigurer注册自定义DatePropertyEditor到容器

```
<bean class="org.springframework.beans.factory.config.CustomEditorConfigurer">
  <property name="customEditors">
    <map>
      <entry key="java.util.Date">
        <ref bean="datePropertyEditor"/>
      </entry>
    </map>
  </property>
</bean>

<bean id="datePropertyEditor" class="...DatePropertyEditor">
  <property name="datePattern">
    <value>yyyy/MM/dd</value>
  </property>
</bean>
```

Spring 2.0之前通常是通过CustomEditorConfigurer的customEditors属性来指定自定义的PropertyEditor。2.0之后，比较提倡使用propertyEditorRegistrars属性来指定自定义的PropertyEditor。不过，这样我们就需要再多做一步工作，就是给出一个org.springframework.beans.PropertyEditorRegistrar的实现。这也很简单，代码清单4-47给出了相应的实例。

代码清单4-47　DatePropertyEditorRegistrar定义

```
public class DatePropertyEditorRegistrar implements PropertyEditorRegistrar {
  private PropertyEditor propertyEditor;

  public void registerCustomEditors(PropertyEditorRegistry peRegistry) {
    peRegistry.registerCustomEditor(java.util.Date.class, getPropertyEditor());
  }

  public PropertyEditor getPropertyEditor() {
    return propertyEditor;
  }

  public void setPropertyEditor(PropertyEditor propertyEditor) {
    this.propertyEditor = propertyEditor;
```

```
    }
}
```

这样，2.0之后所提倡的注册自定义PropertyEditor的方式，如代码清单4-48所示。

代码清单4-48　通过CustomEditorConfigurer的propertyEditorRegistrars注册自定义PropertyEditor

```
<bean class="org.springframework.beans.factory.config.CustomEditorConfigurer">
  <property name="propertyEditorRegistrars">
    <list>
      <ref bean="datePropertyEditorRegistrar"/>
    </list>
  </property>
</bean>

<bean id="datePropertyEditorRegistrar" class="...DatePropertyEditorRegistrar">
  <property name="propertyEditor">
    <ref bean="datePropertyEditor"/>
  </property>
</bean>

<bean id="datePropertyEditor" class="...DatePropertyEditor">
  <property name="datePattern">
    <value>yyyy/MM/dd</value>
  </property>
</bean>
```

要是还有其他扩展类型的PropertyEditor，可以在propertyEditorRegistrars的<list>中一并指定。

4.4.3　了解 bean 的一生

在已经可以借助于BeanFactoryPostProcessor来干预Magic实现的第一个阶段（容器启动阶段）的活动之后，我们就可以开始探索下一个阶段，即bean实例化阶段的实现逻辑了。

容器启动之后，并不会马上就实例化相应的bean定义。我们知道，容器现在仅仅拥有所有对象的BeanDefinition来保存实例化阶段将要用的必要信息。只有当请求方通过BeanFactory的getBean()方法来请求某个对象实例的时候，才有可能触发Bean实例化阶段的活动。BeanFactory的getBean方法可以被客户端对象显式调用，也可以在容器内部隐式地被调用。隐式调用有如下两种情况。

- 对于BeanFactory来说，对象实例化默认采用延迟初始化。通常情况下，当对象A被请求而需要第一次实例化的时候，如果它所依赖的对象B之前同样没有被实例化，那么容器会先实例化对象A所依赖的对象。这时容器内部就会首先实例化对象B，以及对象 A依赖的其他还没有被实例化的对象。这种情况是容器内部调用getBean()，对于本次请求的请求方是隐式的。
- ApplicationContext启动之后会实例化所有的bean定义，这个特性在本书中已经多次提到。但ApplicationContext在实现的过程中依然遵循Spring容器实现流程的两个阶段，只不过它会在启动阶段的活动完成之后，紧接着调用注册到该容器的所有bean定义的实例化方法getBean()。这就是为什么当你得到ApplicationContext类型的容器引用时，容器内所有对象已经被全部实例化完成。不信你查一下类org.springframework.context.support.AbstractApplicationContext的refresh()方法。

之所以说getBean()方法是有可能触发Bean实例化阶段的活动，是因为只有当对应某个bean定义的getBean()方法第一次被调用时，不管是显式的还是隐式的，Bean实例化阶段的活动才会被触发，

第二次被调用则会直接返回容器缓存的第一次实例化完的对象实例（prototype类型bean除外）。当getBean()方法内部发现该bean定义之前还没有被实例化之后，会通过createBean()方法来进行具体的对象实例化，实例化过程如图4-10所示。

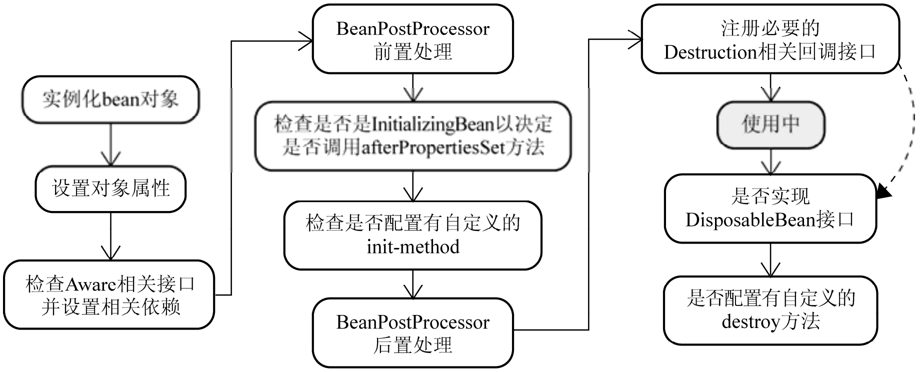

图4-10　Bean的实例化过程

Spring容器将对其所管理的对象全部给予统一的生命周期管理，这些被管理的对象完全摆脱了原来那种"new完后被使用，脱离作用域后即被回收"的命运。下面我们将详细看一看现在的每个bean在容器中是如何走过其一生的。

> **提示**　可以在org.springframework.beans.factory.support.AbstractBeanFactory类的代码中查看到getBean()方法的完整实现逻辑，可以在其子类org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory的代码中一窥createBean()方法的全貌。

1. Bean的实例化与BeanWrapper

容器在内部实现的时候，采用"策略模式（Strategy Pattern）"来决定采用何种方式初始化bean实例。通常，可以通过反射或者CGLIB动态字节码生成来初始化相应的bean实例或者动态生成其子类。

org.springframework.beans.factory.support.InstantiationStrategy定义是实例化策略的抽象接口，其直接子类SimpleInstantiationStrategy实现了简单的对象实例化功能，可以通过反射来实例化对象实例，但不支持方法注入方式的对象实例化。CglibSubclassingInstantiation-Strategy继承了SimpleInstantiationStrategy的以反射方式实例化对象的功能，并且通过CGLIB的动态字节码生成功能，该策略实现类可以动态生成某个类的子类，进而满足了方法注入所需的对象实例化需求。默认情况下，容器内部采用的是CglibSubclassingInstantiationStrategy。

容器只要根据相应bean定义的BeanDefintion取得实例化信息，结合CglibSubclassingIns-tantiationStrategy以及不同的bean定义类型，就可以返回实例化完成的对象实例。但是，返回方式上有些"点缀"。不是直接返回构造完成的对象实例，而是以BeanWrapper对构造完成的对象实例进行包裹，返回相应的BeanWrapper实例。

至此，第一步结束。

BeanWrapper接口通常在Spring框架内部使用，它有一个实现类org.springframework.beans.BeanWrapperImpl。其作用是对某个bean进行"包裹"，然后对这个"包裹"的bean进行操作，比如

设置或者获取bean的相应属性值。而在第一步结束后返回BeanWrapper实例而不是原先的对象实例，就是为了第二步“设置对象属性”。

BeanWrapper定义继承了org.springframework.beans.PropertyAccessor接口，可以以统一的方式对对象属性进行访问；BeanWrapper定义同时又直接或者间接继承了PropertyEditorRegistry和TypeConverter接口。不知你是否还记得CustomEditorConfigurer？当把各种PropertyEditor注册给容器时，知道后面谁用到这些PropertyEditor吗？对，就是BeanWrapper！在第一步构造完成对象之后，Spring会根据对象实例构造一个BeanWrapperImpl实例，然后将之前CustomEditor-Configurer注册的PropertyEditor复制一份给BeanWrapperImpl实例（这就是BeanWrapper同时又是PropertyEditorRegistry的原因）。这样，当BeanWrapper转换类型、设置对象属性值时，就不会无从下手了。

使用BeanWrapper对bean实例操作很方便，可以免去直接使用Java反射API（Java Reflection API）操作对象实例的烦琐。来看一段代码（见代码清单4-49），之后我们就会更加清楚Spring容器内部是如何设置对象属性的了！

代码清单4-49　使用BeanWrapper操作对象

```
Object provider = Class.forName("package.name.FXNewsProvider").newInstance();
Object listener = Class.forName("package.name.DowJonesNewsListener").newInstance();
Object persister = Class.forName("package.name.DowJonesNewsPersister").newInstance();

BeanWrapper newsProvider = new BeanWrapperImpl(provider);
newsProvider.setPropertyValue("newsListener", listener);
newsProvider.setPropertyValue("newPersistener", persister);

assertTrue(newsProvider.getWrappedInstance() instanceof FXNewsProvider);
assertSame(provider, newsProvider.getWrappedInstance());
assertSame(listener, newsProvider.getPropertyValue("newsListener"));
assertSame(persister, newsProvider.getPropertyValue("newPersistener"));
```

我想有了BeanWrapper的帮助，你不会想直接使用Java反射API来做同样事情的。代码清单4-50演示了同样的功能，即直接使用Java反射API是如何实现的（忽略了异常处理相关代码）。

代码清单4-50　直接使用Java反射API操作对象

```
Object provider = Class.forName("package.name.FXNewsProvider").newInstance();
Object listener = Class.forName("package.name.DowJonesNewsListener").newInstance();
Object persister = Class.forName("package.name.DowJonesNewsPersister").newInstance();

Class providerClazz = provider.getClass();
Field listenerField = providerClazz.getField("newsListener");
listenerField.set(provider, listener);
Field persisterField = providerClazz.getField("newsListener");
persisterField.set(provider, persister);

assertSame(listener, listenerField.get(provider));
assertSame(persister, persisterField.get(provider));
```

如果你觉得没有太大差别，那是因为没有看到紧随其后的那些异常（exception）还有待处理！

2. 各色的Aware接口

当对象实例化完成并且相关属性以及依赖设置完成之后，Spring容器会检查当前对象实例是否实现了一系列的以Aware命名结尾的接口定义。如果是，则将这些Aware接口定义中规定的依赖注入给

当前对象实例。

这些Aware接口为如下几个。

- org.springframework.beans.factory.BeanNameAware。如果Spring容器检测到当前对象实例实现了该接口，会将该对象实例的bean定义对应的beanName设置到当前对象实例。
- org.springframework.beans.factory.BeanClassLoaderAware。如果容器检测到当前对象实例实现了该接口，会将对应加载当前bean的Classloader注入当前对象实例。默认会使用加载org.springframework.util.ClassUtils类的Classloader。
- org.springframework.beans.factory.BeanFactoryAware。在介绍方法注入的时候，我们提到过使用该接口以便每次获取prototype类型bean的不同实例。如果对象声明实现了BeanFactoryAware接口，BeanFactory容器会将自身设置到当前对象实例。这样，当前对象实例就拥有了一个BeanFactory容器的引用，并且可以对这个容器内允许访问的对象按照需要进行访问。

以上几个Aware接口只是针对BeanFactory类型的容器而言，对于ApplicationContext类型的容器，也存在几个Aware相关接口。不过在检测这些接口并设置相关依赖的实现机理上，与以上几个接口处理方式有所不同，使用的是下面将要说到的BeanPostProcessor方式。不过，设置Aware接口这一步与BeanPostProcessor是相邻的，把这几个接口放到这里一起提及，也没什么不可以的。

对于ApplicationContext类型容器，容器在这一步还会检查以下几个Aware接口并根据接口定义设置相关依赖。

- org.springframework.context.ResourceLoaderAware。ApplicationContext实现了Spring的ResourceLoader接口（后面会提及详细信息）。当容器检测到当前对象实例实现了ResourceLoaderAware接口之后，会将当前ApplicationContext自身设置到对象实例，这样当前对象实例就拥有了其所在ApplicationContext容器的一个引用。
- org.springframework.context.ApplicationEventPublisherAware。ApplicationContext作为一个容器，同时还实现了ApplicationEventPublisher接口，这样，它就可以作为ApplicationEventPublisher来使用。所以，当前ApplicationContext容器如果检测到当前实例化的对象实例声明了ApplicationEventPublisherAware接口，则会将自身注入当前对象。
- org.springframework.context.MessageSourceAware。ApplicationContext通过MessageSource接口提供国际化的信息支持，即I18n（Internationalization）。它自身就实现了MessageSource接口，所以当检测到当前对象实例实现了MessageSourceAware接口，则会将自身注入当前对象实例。
- org.springframework.context.ApplicationContextAware。如果ApplicationContext容器检测到当前对象实现了ApplicationContextAware接口，则会将自身注入当前对象实例。

3. **BeanPostProcessor**

BeanPostProcessor的概念容易与BeanFactoryPostProcessor的概念混淆。但只要记住BeanPostProcessor是存在于对象实例化阶段，而BeanFactoryPostProcessor则是存在于容器启动阶段，这两个概念就比较容易区分了。

与BeanFactoryPostProcessor通常会处理容器内所有符合条件的BeanDefinition类似，BeanPostProcessor会处理容器内所有符合条件的实例化后的对象实例。该接口声明了两个方法，分别在两个不同的时机执行，见如下代码定义：

```
public interface BeanPostProcessor
```

```
{
  Object postProcessBeforeInitialization(Object bean, String beanName) throws ➥
  BeansException;
  Object postProcessAfterInitialization(Object bean, String beanName) throws ➥
  BeansException;
}
```

postProcessBeforeInitialization()方法是图4-10中BeanPostProcessor前置处理这一步将会执行的方法，postProcessAfterInitialization()则是对应图4-10中BeanPostProcessor后置处理那一步将会执行的方法。BeanPostProcessor的两个方法中都传入了原来的对象实例的引用，这为我们扩展容器的对象实例化过程中的行为提供了极大的便利，我们几乎可以对传入的对象实例执行任何的操作。

通常比较常见的使用BeanPostProcessor的场景，是处理标记接口实现类，或者为当前对象提供代理实现。在图4-10的第三步中，ApplicationContext对应的那些Aware接口实际上就是通过BeanPostProcessor的方式进行处理的。当ApplicationContext中每个对象的实例化过程走到BeanPostProcessor前置处理这一步时，ApplicationContext容器会检测到之前注册到容器的ApplicationContextAwareProcessor这个BeanPostProcessor的实现类，然后就会调用其postProcessBeforeInitialization()方法，检查并设置Aware相关依赖。ApplicationContextAwareProcessor的postProcessBeforeInitialization()代码很简单明了，见代码清单4-51。

代码清单4-51　postProcessBeforeInitialization方法定义

```
public Object postProcessBeforeInitialization(Object bean, String beanName) throws ➥
BeansException {
  if (bean instanceof ResourceLoaderAware) {
    ((ResourceLoaderAware) bean).setResourceLoader(this.applicationContext);
  }
  if (bean instanceof ApplicationEventPublisherAware) {
    ((ApplicationEventPublisherAware) bean).setApplicationEventPublisher ➥
    (this.applicationContext);
  }
  if (bean instanceof MessageSourceAware) {
    ((MessageSourceAware) bean).setMessageSource(this.applicationContext);
  }
  if (bean instanceof ApplicationContextAware) {
    ((ApplicationContextAware) bean).setApplicationContext(this.applicationContext);
  }
  return bean;
}
```

除了检查标记接口以便应用自定义逻辑，还可以通过BeanPostProcessor对当前对象实例做更多的处理。比如替换当前对象实例或者字节码增强当前对象实例等。Spring的AOP则更多地使用BeanPostProcessor来为对象生成相应的代理对象，如org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator。我们将在Spring AOP部分详细介绍该类和AOP相关概念。

BeanPostProcessor是容器提供的对象实例化阶段的强有力的扩展点。为了进一步演示它的强大威力，我们有必要实现一个自定义的BeanPostProcessor。

- **自定义BeanPostProcessor**

假设系统中所有的IFXNewsListener实现类需要从某个位置取得相应的服务器连接密码，而且系统中保存的密码是加密的，那么在IFXNewsListener发送这个密码给新闻服务器进行连接验证的时

候，首先需要对系统中取得的密码进行解密，然后才能发送。我们将采用BeanPostProcessor技术，对所有的IFXNewsListener的实现类进行统一的解密操作。

(1) 标注需要进行解密的实现类

为了能够识别那些需要对服务器连接密码进行解密的IFXNewsListener实现，我们声明了接口PasswordDecodable，并要求相关IFXNewsListener实现类实现该接口。PasswordDecodable接口声明以及相关的IFXNewsListener实现类定义见代码清单4-52。

代码清单4-52　PasswordDecodable接口声明以及相关的IFXNewsListener实现类

```
public interface PasswordDecodable {
  String getEncodedPassword();
  void setDecodedPassword(String password);
}

public class DowJonesNewsListener implements IFXNewsListener,PasswordDecodable {
  private String password;

  public String[] getAvailableNewsIds() {
    // 省略
  }

  public FXNewsBean getNewsByPK(String newsId) {
    // 省略
  }

  public void postProcessIfNecessary(String newsId) {
    // 省略
  }

  public String getEncodedPassword() {
    return this.password;
  }

  public void setDecodedPassword(String password) {
    this.password = password;
  }

}
```

(2) 实现相应的BeanPostProcessor对符合条件的Bean实例进行处理

我们通过PasswordDecodable接口声明来区分将要处理的对象实例[1]，当检查到当前对象实例实现了该接口之后，就会从当前对象实例取得加密后的密码，并对其解密。然后将解密后的密码设置回当前对象实例。之后，返回的对象实例所持有的就是解密后的密码，逻辑如代码清单4-53所示。

代码清单4-53　用于解密的自定义BeanPostProcessor实现类

```
public class PasswordDecodePostProcessor implements BeanPostProcessor {

  public Object postProcessAfterInitialization(Object object, String beanName)
      throws BeansException {
    return object;
  }
  public Object postProcessBeforeInitialization(Object object, String beanName)
      throws BeansException {
    if(object instanceof PasswordDecodable)
```

① 如果有其他方式可以区分将要处理的对象实例，那么声明类似的标记接口（Marker Interface）就不是必须的。

```
        {
            String encodedPassword = ((PasswordDecodable)object).getEncodedPassword();
            String decodedPassword = decodePassword(encodedPassword);
            ((PasswordDecodable)object).setDecodedPassword(decodedPassword);
        }
        return object;
    }
    private String decodePassword(String encodedPassword) {
        // 实现解码逻辑
        return encodedPassword;
    }
}
```

(3) 将自定义的BeanPostProcessor注册到容器

只有将自定义的BeanPostProcessor实现类告知容器，容器才会在合适的时机应用它。所以，我们需要将PasswordDecodePostProcessor注册到容器。

对于BeanFactory类型的容器来说，我们需要通过手工编码的方式将相应的BeanPostProcessor注册到容器，也就是调用ConfigurableBeanFactory的addBeanPostProcessor()方法，见如下代码：

```
ConfigurableBeanFactory beanFactory = new XmlBeanFactory(new ClassPathResource(...));
beanFactory.addBeanPostProcessor(new PasswordDecodePostProcessor());
...
// getBean();
```

对于ApplicationContext容器来说，事情则方便得多，直接将相应的BeanPostProcessor实现类通过通常的XML配置文件配置一下即可。ApplicationContext容器会自动识别并加载注册到容器的BeanPostProcessor，如下配置内容将我们的PasswordDecodePostProcessor注册到容器：

```
<beans>
    <bean id="passwordDecodePostProcessor" class="package.name.PasswordDecodePostProcessor">
        <!--如果需要，注入必要的依赖-->
    </bean>

    ...
</beans>
```

合理利用BeanPostProcessor这种Spring的容器扩展机制，将可以构造强大而灵活的应用系统。

提示 实际上，有一种特殊类型的BeanPostProcessor我们没有提到，它的执行时机与通常的BeanPostProcessor不同。org.springframework.beans.factory.config.InstantiationAwareBeanPostProcessor接口可以在对象的实例化过程中导致某种类似于电路“短路”的效果。实际上，并非所有注册到Spring容器内的bean定义都是按照图4-10的流程实例化的。在所有的步骤之前，也就是实例化bean对象步骤之前，容器会首先检查容器中是否注册有InstantiationAwareBeanPostProcessor类型的BeanPostProcessor。如果有，首先使用相应的InstantiationAwareBeanPostProcessor来构造对象实例。构造成功后直接返回构造完成的对象实例，而不会按照“正规的流程”继续执行。这就是它可能造成“短路”的原因。

不过，通常情况下都是Spring容器内部使用这种特殊类型的BeanPostProcessor做一些动态对象代理等工作，我们使用普通的BeanPostProcessor实现就可以。这里简单提及一下，目的是让大家有所了解。

4. InitializingBean和init-method

org.springframework.beans.factory.InitializingBean是容器内部广泛使用的一个对象生命周期标识接口，其定义如下：

```
public interface InitializingBean {
    void afterPropertiesSet() throws Exception;
}
```

该接口定义很简单，其作用在于，在对象实例化过程调用过“BeanPostProcessor的前置处理”之后，会接着检测当前对象是否实现了InitializingBean接口，如果是，则会调用其afterPropertiesSet()方法进一步调整对象实例的状态。比如，在有些情况下，某个业务对象实例化完成后，还不能处于可以使用状态。这个时候就可以让该业务对象实现该接口，并在方法afterPropertiesSet()中完成对该业务对象的后续处理。

虽然该接口在Spring容器内部广泛使用，但如果真的让我们的业务对象实现这个接口，则显得Spring容器比较具有侵入性。所以，Spring还提供了另一种方式来指定自定义的对象初始化操作，那就是在XML配置的时候，使用<bean>的init-method属性。

通过init-method，系统中业务对象的自定义初始化操作可以以任何方式命名，而不再受制于InitializingBean的afterPropertiesSet()。如果系统开发过程中规定：所有业务对象的自定义初始化操作都必须以init()命名，为了省去挨个<bean>的设置init-method这样的烦琐，我们还可以通过最顶层的<beans>的default-init-method统一指定这一init()方法名。

一般，我们是在集成第三方库，或者其他特殊的情况下，才会需要使用该特性。比如，ObjectLab提供了一个外汇系统交易日计算的开源实现——ObjectLabKit，系统在使用它提供的DateCalculator时，封装类会通过一个自定义的初始化方法来为这些DateCalculator提供计算交易日所需要排除的休息日信息。代码清单4-54给出了封装类的部分代码。

代码清单4-54　DateCalculator封装类定义

```
public class FXTradeDateCalculator {
    public  static final DateTimeFormatter FRONT_DATE_FORMATTER = ➥
    DateTimeFormat.forPattern("yyyyMMdd");
    private static final Set<LocalDate> holidaySet = ➥
    new HashSet<LocalDate>();
    private static final String holidayKey = "JPY";
    private SqlMapClientTemplate sqlMapClientTemplate;

    public FXTradeDateCalculator(SqlMapClientTemplate sqlMapClientTemplate)
    {
        this.sqlMapClientTemplate = sqlMapClientTemplate;
    }

    public void setupHolidays()
    {
        List holidays = getSystemHolidays();
        if(!ListUtils.isEmpty(holidays))
        {
            for(int i=0,size=holidays.size();i<size;i++)
            {
                String holiday = (String)holidays.get(i);
                LocalDate date = ➥
                FRONT_DATE_FORMATTER.parseDateTime(holiday).toLocalDate();
```

```
                holidaySet.add(date);
            }
        }
        LocalDateKitCalculatorsFactory ➥
        .getDefaultInstance().registerHolidays(holidayKey, holidaySet);
    }
    public DateCalculator<LocalDate> getForwardDateCalculator()
    {
        return LocalDateKitCalculatorsFactory ➥
            .getDefaultInstance() ➥
            .getDateCalculator(holidayKey, HolidayHandlerType.FORWARD);
        }

    public DateCalculator<LocalDate> getBackwardDateCalculator()
    {
        return LocalDateKitCalculatorsFactory ➥
            .getDefaultInstance() ➥
            .getDateCalculator(holidayKey, HolidayHandlerType.BACKWARD);
    }
    public List getSystemHolidays()
    {
        return getSqlMapClientTemplate() ➥
        .queryForList("CommonContext.holiday", null);
    }
}
```

为了保证getForwardDateCalculator()和getBackwardDateCalculator()方法返回的DateCalculator已经将休息日考虑进去，在这两个方法被调用之前，我们需要setupHolidays()首先被调用，以保证将休息日告知DateCalculator，使它能够在计算交易日的时候排除掉这些休息日的日期。因此，我们需要在配置文件中完成类似代码清单4-55所示的配置，以保证在对象可用之前，setupHolidays()方法会首先被调用。

代码清单4-55 使用init-method保证封装类的初始化方法得以执行

```
<beans>
  <bean id="tradeDateCalculator" class="FXTradeDateCalculator" ➥
  init-method="setupHolidays">
    <constructor-arg>
      <ref bean="sqlMapClientTemplate"/>
    </constructor-arg>
  </bean>

  <bean id="sqlMapClientTemplate" ➥
  class="org.springframework.orm.ibatis.SqlMapClientTemplate">
    ...
  </bean>

  ...
</beans>
```

当然，我们也可以让FXTradeDateCalculator实现InitializingBean接口，然后将setupHolidays()方法的逻辑转移到afterPropertiesSet()方法。不过，相对来说还是采用init-method的方式比较灵活，并且没有那么强的侵入性。

可以认为在InitializingBean和init-method中任选其一就可以帮你完成类似的初始化工作。除非……，除非你真的那么“幸运”，居然需要在同一个业务对象上按照先后顺序执行两个初始化方

法。这个时候，就只好在同一对象上既实现InitializingBean的afterPropertiesSet()，又提供自定义初始化方法啦！

5. DisposableBean与destroy-method

当所有的一切，该设置的设置，该注入的注入，该调用的调用完成之后，容器将检查singleton类型的bean实例，看其是否实现了org.springframework.beans.factory.DisposableBean接口。或者其对应的bean定义是否通过<bean>的destroy-method属性指定了自定义的对象销毁方法。如果是，就会为该实例注册一个用于对象销毁的回调（Callback），以便在这些singleton类型的对象实例销毁之前，执行销毁逻辑。

与InitializingBean和init-method用于对象的自定义初始化相对应，DisposableBean和destroy-method为对象提供了执行自定义销毁逻辑的机会。

最常见到的该功能的使用场景就是在Spring容器中注册数据库连接池，在系统退出后，连接池应该关闭，以释放相应资源。代码清单4-56演示了通常情况下使用destroy-method处理资源释放的数据源注册配置。

代码清单4-56　使用了自定义销毁方法的数据源配置定义

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" ➥
destroy-method="close">
  <property name="url">
    <value>${jdbc.url}</value>
  </property>
  <property name="driverClassName">
    <value>${jdbc.driver}</value>
  </property>
  <property name="username">
    <value>${jdbc.username}</value>
  </property>
  <property name="password">
    <value>${jdbc.password}</value>
  </property>
  ...
</bean>
```

不过，这些自定义的对象销毁逻辑，在对象实例初始化完成并注册了相关的回调方法之后，并不会马上执行。回调方法注册后，返回的对象实例即处于使用状态，只有该对象实例不再被使用的时候，才会执行相关的自定义销毁逻辑，此时通常也就是Spring容器关闭的时候。但Spring容器在关闭之前，不会聪明到自动调用这些回调方法。所以，需要我们告知容器，在哪个时间点来执行对象的自定义销毁方法。

对于BeanFactory容器来说。我们需要在独立应用程序的主程序退出之前，或者其他被认为是合适的情况下（依照应用场景而定），如代码清单4-57所示，调用ConfigurableBeanFactory提供的destroySingletons()方法销毁容器中管理的所有singleton类型的对象实例。

代码清单4-57　使用ConfigurableBeanFactory的destroySingletons()方法触发销毁对象行为

```
public class ApplicationLauncher
{
  public static void main(String[] args) {
  BasicConfigurator.configure();
  BeanFactory container = new XmlBeanFactory(new ClassPathResource("..."));
  BusinessObject bean = (BusinessObject)container.getBean("...");
```

```
    bean.doSth();
    ((ConfigurableListableBeanFactory)container).destroySingletons();
    // 应用程序退出，容器关闭
    }
}
```

如果不能在合适的时机调用destroySingletons()，那么所有实现了DisposableBean接口的对象实例或者声明了destroy-method的bean定义对应的对象实例，它们的自定义对象销毁逻辑就形同虚设，因为根本就不会被执行！

对于ApplicationContext容器来说。道理是一样的。但AbstractApplicationContext为我们提供了registerShutdownHook()方法，该方法底层使用标准的Runtime类的addShutdownHook()方式来调用相应bean对象的销毁逻辑，从而保证在Java虚拟机退出之前，这些singtleton类型的bean对象实例的自定义销毁逻辑会被执行。当然AbstractApplicationContext注册的shutdownHook不只是调用对象实例的自定义销毁逻辑，也包括ApplicationContext相关的事件发布等，代码清单4-58演示了该方法的使用。

代码清单4-58 使用registerShutdownHook()方法注册并触发对象销毁逻辑回调行为

```
public class ApplicationLauncher
{
  public static void main(String[] args) {
    BasicConfigurator.configure();
    BeanFactory container = new ClassPathXmlApplicationContext("...");
    ((AbstractApplicationContext)container).registerShutdownHook();
    BusinessObject bean = (BusinessObject)container.getBean("...");
    bean.doSth();
    // 应用程序退出，容器关闭
  }
}
```

同样的道理，在Spring 2.0引入了自定义scope之后，使用自定义scope的相关对象实例的销毁逻辑，也应该在合适的时机被调用执行。不过，所有这些规则不包含prototype类型的bean实例，因为prototype对象实例在容器实例化并返回给请求方之后，容器就不再管理这种类型对象实例的生命周期了。

至此，bean走完了它在容器中“光荣”的一生。

4.5 小结

Spring的IoC容器主要有两种，即BeanFactory和ApplicationContext。本章伊始，首先对这两种容器做了总体上的介绍，然后转入本章的重点，也就是Spring的BeanFactory基础容器。

我们从对比使用BeanFactory开发前后的差别开始，阐述了BeanFactory作为一个具体的IoC Service Provider，它是如何支持各种对象注册以及依赖关系绑定的。XML自始至终都是Spring的IoC容器支持最完善的Configuration Metadata提供方式。所以，我们接着从XML入手，深入挖掘了BeanFactory（以及ApplicationContext）的各种潜力。

对于充满好奇心的我们，不会只停留在会使用BeanFactory进行开发这一层面。所以，最后我们又一起探索了BeanFactory（当然，也是ApplicationContext）实现背后的各种奥秘。BeanFactory是Spring提供的基础IoC容器，但并不是Spring提供的唯一IoC容器。ApplicationContext构建于BeanFactory之上，提供了许多BeanFactory之外的特性。下一章，我们将一起走入Application-Context的世界。

第5章 Spring IoC容器 ApplicationContext

本章内容

- 统一资源加载策略
- 国际化信息支持
- 容器内部事件发布
- 多配置模块加载的简化

作为Spring提供的较之BeanFactory更为先进的IoC容器实现，ApplicationContext除了拥有BeanFactory支持的所有功能之外，还进一步扩展了基本容器的功能，包括BeanFactoryPostProcessor、BeanPostProcessor以及其他特殊类型bean的自动识别、容器启动后bean实例的自动初始化、国际化的信息支持、容器内事件发布等。真是“青出于蓝而胜于蓝”啊！

Spring为基本的BeanFactory类型容器提供了XmlBeanFactory实现。相应地，它也为ApplicationContext类型容器提供了以下几个常用的实现。

- org.springframework.context.support.FileSystemXmlApplicationContext。在默认情况下，从文件系统加载bean定义以及相关资源的ApplicationContext实现。
- org.springframework.context.support.ClassPathXmlApplicationContext。在默认情况下，从Classpath加载bean定义以及相关资源的ApplicationContext实现。
- org.springframework.web.context.support.XmlWebApplicationContext。Spring提供的用于Web应用程序的ApplicationContext实现，我们将在第六部分更多地接触到它。

更多实现可以参照org.springframework.context.ApplicationContext接口定义的Javadoc，这里不再赘述。

第4章中说明了ApplicationContext所支持的大部分功能。下面主要围绕ApplicationContext较之BeanFactory特有的一些特性展开讨论，即国际化（I18n）信息支持、统一资源加载策略以及容器内事件发布等。

5.1 统一资源加载策略

要搞清楚Spring为什么提供这么一个功能，还是从Java SE提供的标准类java.net.URL说起比较好。URL全名是Uniform Resource Locator（统一资源定位器），但多少有些名不副实的味道。

首先，说是统一资源定位，但基本实现却只限于网络形式发布的资源的查找和定位工作，基本上只提供了基于HTTP、FTP、File等协议（sun.net.www.protocol包下所支持的协议）的资源定位功能。虽然也提供了扩展的接口，但从一开始，其自身的“定位”就已经趋于狭隘了。实际上，资源这个词的范围比较广义，资源可以任何形式存在，如以二进制对象形式存在、以字节流形式存在、以文件形式存在等；而且，资源也可以存在于任何场所，如存在于文件系统、存在于Java应用的Classpath中，甚至存在于URL可以定位的地方。

其次，从某些程度上来说，该类的功能职责划分不清，资源的查找和资源的表示没有一个清晰的界限。当前情况是，资源查找后返回的形式多种多样，没有一个统一的抽象。理想情况下，资源查找完成后，返回给客户端的应该是一个统一的资源抽象接口，客户端要对资源进行什么样的处理，应该由资源抽象接口来界定，而不应该成为资源的定位者和查找者同时要关心的事情。

所以，在这个前提下①，Spring提出了一套基于org.springframework.core.io.Resource和org.springframework.core.io.ResourceLoader接口的资源抽象和加载策略。

5.1.1 Spring中的Resource

Spring框架内部使用org.springframework.core.io.Resource接口作为所有资源的抽象和访问接口，我们之前在构造BeanFactory的时候已经接触过它，如下代码：

```
BeanFactory beanFactory = new XmlBeanFactory(new ClassPathResource("..."));
...
```

其中ClassPathResource就是Resource的一个特定类型的实现，代表的是位于Classpath中的资源。

Resource接口可以根据资源的不同类型，或者资源所处的不同场合，给出相应的具体实现。Spring框架在这个理念的基础上，提供了一些实现类（可以在org.springframework.core.io包下找到这些实现类）。

- ByteArrayResource。将字节（byte）数组提供的数据作为一种资源进行封装，如果通过InputStream形式访问该类型的资源，该实现会根据字节数组的数据，构造相应的ByteArray-InputStream并返回。
- ClassPathResource。该实现从Java应用程序的ClassPath中加载具体资源并进行封装，可以使用指定的类加载器（ClassLoader）或者给定的类进行资源加载。
- FileSystemResource。对java.io.File类型的封装，所以，我们可以以文件或者URL的形式对该类型资源进行访问，只要能跟File打的交道，基本上跟FileSystemResource也可以。
- UrlResource。通过java.net.URL进行的具体资源查找定位的实现类，内部委派URL进行具体的资源操作。
- InputStreamResource。将给定的InputStream视为一种资源的Resource实现类，较为少用。可能的情况下，以ByteArrayResource以及其他形式资源实现代之。

如果以上这些资源实现还不能满足要求，那么我们还可以根据相应场景给出自己的实现，只需实现org.springframework.core.io.Resource接口就是了。代码清单5-1给出了该接口的定义。

代码清单5-1 Resource接口定义

```
public interface Resource extends InputStreamSource {

  boolean exists();

  boolean isOpen();

  URL getURL() throws IOException;

  File getFile() throws IOException;
```

① 只是个人看法，至于说SpringTeam基于如何考虑，我也不知，或许只是给出了URL之外的另一种选择，也可能是“重新发明一套更好用的轮子”，我姑且说之，你姑且听之，或者用之……

```
    Resource createRelative(String relativePath) throws IOException;

    String getFilename();

    String getDescription();

}

public interface InputStreamSource {
    InputStream getInputStream() throws IOException;
}
```

该接口定义了7个方法，可以帮助我们查询资源状态、访问资源内容，甚至根据当前资源创建新的相对资源。不过，要真想实现自定义的Resource，倒是真没必要直接实现该接口，我们可以继承org.springframework.core.io.AbstractResource抽象类，然后根据当前具体资源特征，覆盖相应的方法就可以了。什么？让我给个实现的例子？算了吧，目前我还没碰到这样的需求。呵呵！要真的碰上了，你只要知道有这么“一出儿”就行了。

5.1.2　ResourceLoader，“更广义的URL”

资源是有了，但如何去查找和定位这些资源，则应该是ResourceLoader的职责所在了。org.springframework.core.io.ResourceLoader接口是资源查找定位策略的统一抽象，具体的资源查找定位策略则由相应的ResourceLoader实现类给出。我想，把ResourceLoader称作统一资源定位器或许才更恰当一些吧！ResourceLoader定义如下：

```
public interface ResourceLoader {
    String CLASSPATH_URL_PREFIX = ResourceUtils.CLASSPATH_URL_PREFIX;
    Resource getResource(String location);
    ClassLoader getClassLoader();
}
```

其中最主要的就是Resource getResource(String location);方法，通过它，我们就可以根据指定的资源位置，定位到具体的资源实例。

1. 可用的ResourceLoader

- DefaultResourceLoader

ResourceLoader有一个默认的实现类，即org.springframework.core.io.DefaultResource-Loader，该类默认的资源查找处理逻辑如下。

(1) 首先检查资源路径是否以classpath:前缀打头，如果是，则尝试构造ClassPathResource类型资源并返回。

(2) 否则，(a) 尝试通过URL，根据资源路径来定位资源，如果没有抛出MalformedURLException，有则会构造UrlResource类型的资源并返回；(b)如果还是无法根据资源路径定位指定的资源，则委派getResourceByPath(String)方法来定位，DefaultResourceLoader的getResourceByPath(String)方法默认实现逻辑是，构造ClassPathResource类型的资源并返回。

在这个基础上，让我们来看一下DefaultResourceLoader的行为是如何反应到程序中的吧！代码清单5-2给出的代码片段演示了DefaultResourceLoader的具体行为。

代码清单5-2　DefaultResourceLoader使用演示

```
ResourceLoader resourceLoader = new DefaultResourceLoader();
```

```
Resource fakeFileResource = resourceLoader.getResource("D:/spring21site/README");
assertTrue(fakeFileResource instanceof ClassPathResource);
assertFalse(fakeFileResource.exists());

Resource urlResource1 = resourceLoader.getResource("file:D:/spring21site/README");
assertTrue(urlResource1 instanceof UrlResource);

Resource urlResource2 = resourceLoader.getResource("http://www.spring21.cn");
assertTrue(urlResource2 instanceof UrlResource);

try{
  fakeFileResource.getFile();
  fail("no such file with path["+fakeFileResource.getFilename()+"] exists in classpath");
}
catch(FileNotFoundException e){
  //
}
try{
  urlResource1.getFile();
}
catch(FileNotFoundException e){
  fail();
}
```

尤其注意fakeFileResource资源的类型，并不是我们所预期的FileSystemResource类型，而是ClassPathResource类型，这是由DefaultResourceLoader的资源查找逻辑所决定的。如果最终没有找到符合条件的相应资源，getResourceByPath(String)方法就会构造一个实际上并不存在的资源并返回。而指定有协议前缀的资源路径，则通过URL能够定位，所以，返回的都是UrlResource类型。

- FileSystemResourceLoader

为了避免DefaultResourceLoader在最后getResourceByPath(String)方法上的不恰当处理，我们可以使用org.springframework.core.io.FileSystemResourceLoader，它继承自Default-ResourceLoader，但覆写了getResourceByPath(String)方法，使之从文件系统加载资源并以FileSystemResource类型返回。这样，我们就可以取得预想的资源类型。代码清单5-3中的代码将帮助我们验证这一点。

代码清单5-3　使用FileSystemResourceLoader

```
public void testResourceTypesWithFileSystemResourceLoader()
{
    ResourceLoader resourceLoader = new FileSystemResourceLoader();
    Resource fileResource = resourceLoader.getResource("D:/spring21site/README");
    assertTrue(fileResource instanceof FileSystemResource);
    assertTrue(fileResource.exists());

    Resource urlResource = resourceLoader.getResource("file:D:/spring21site/README");
    assertTrue(urlResource instanceof UrlResource);
}
```

FileSystemResourceLoader在ResourceLoader家族中的兄弟FileSystemXmlApplication-Context，也是覆写了getResourceByPath(String)方法的逻辑，以改变DefaultResourceLoader的默认资源加载行为，最终从文件系统中加载并返回FileSystemResource类型的资源。

2. **ResourcePatternResolver——批量查找的ResourceLoader**

ResourcePatternResolver是ResourceLoader的扩展，ResourceLoader每次只能根据资源路径返回确定的单个Resource实例，而ResourcePatternResolver则可以根据指定的资源路径匹配模式，每次返回多个Resource实例。接口org.springframework.core.io.support.ResourcePattern-Resolver定义如下：

```
public interface ResourcePatternResolver extends ResourceLoader {
   String CLASSPATH_ALL_URL_PREFIX = "classpath*:";
   Resource[] getResources(String locationPattern) throws IOException;
}
```

ResourcePatternResolver在继承ResourceLoader原有定义的基础上，又引入了Resource[] getResources(String)方法定义，以支持根据路径匹配模式返回多个Resources的功能。它同时还引入了一种新的协议前缀classpath*:，针对这一点的支持，将由相应的子类实现给出。

ResourcePatternResolver最常用的一个实现是org.springframework.core.io.support.PathMatchingResourcePatternResolver，该实现类支持ResourceLoader级别的资源加载，支持基于Ant风格的路径匹配模式（类似于**/*.suffix之类的路径形式），支持ResourcePatternResolver新增加的classpath*:前缀等，基本上集所有技能于一身。

在构造PathMatchingResourcePatternResolver实例的时候，可以指定一个ResourceLoader，如果不指定的话，则PathMatchingResourcePatternResolver内部会默认构造一个Default-ResourceLoader实例。PathMatchingResourcePatternResolver内部会将匹配后确定的资源路径，委派给它的ResourceLoader来查找和定位资源。这样，如果不指定任何ResourceLoader的话，Path-MatchingResourcePatternResolver在加载资源的行为上会与DefaultResourceLoader基本相同，只存在返回的Resource数量上的差异。如下代码表明了二者在资源加载行为上的一致性：

```
ResourcePatternResolver resourceResolver = new PathMatchingResourcePatternResolver();
Resource fileResource = resourceResolver.getResource("D:/spring21site/README");
assertTrue(fileResource instanceof ClassPathResource);
assertFalse(fileResource.exists());
...
```

不过，可以通过传入其他类型的ResourceLoader来替换PathMatchingResourcePatternResolver内部默认使用的DefaultResourceLoader，从而改变其默认行为。比如，可以如代码清单5-4所示，使用FileSystemResourceLoader替换默认的DefaultResourceLoader，从而使得PathMatching-ResourcePatternResolver的行为跟使用FileSystemResourceLoader一样。

代码清单5-4　替换DefaultResourceLoader后的PathMatchingResourcePatternResolver

```
public void testResourceTypesWithPathMatchingResourcePatternResolver()
{
   ResourcePatternResolver resourceResolver = new PathMatchingResourcePatternResolver();
   Resource fileResource = resourceResolver.getResource("D:/spring21site/README");
   assertTrue(fileResource instanceof ClassPathResource);
   assertFalse(fileResource.exists());

   resourceResolver = new PathMatchingResourcePatternResolver(new ➥
   FileSystemResourceLoader());
   fileResource = resourceResolver.getResource("D:/spring21site/README");
   assertTrue(fileResource instanceof FileSystemResource);
   assertTrue(fileResource.exists());
}
```

3. 回顾与展望

现在我们应该对Spring的统一资源加载策略有了一个整体上的认识，就如图5-1所示。

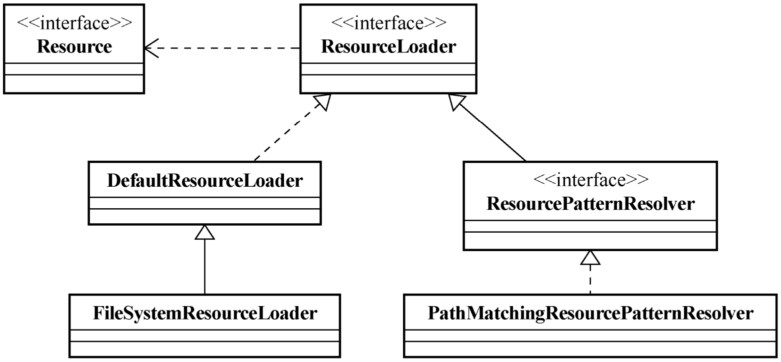

图5-1 Resource和ResourceLoader类层次图

虽然现在看来比较“单薄”，不过，稍后，我们就会发现情况并非如此了。

5.1.3 ApplicationContext与ResourceLoader

说是讲ApplicationContext的统一资源加载策略，到目前为止却一直没有涉及任何ApplicationContext相关的内容，不知道你是否开始奇怪了呢？实际上，我是有意为之，就是不想让各位因为过多关注ApplicationContext，却忽略了事情的本质。

如果回头看一下图4-2，就会发现，ApplicationContext继承了ResourcePatternResolver，当然就间接实现了ResourceLoader接口。所以，任何的ApplicationContext实现都可以看作是一个ResourceLoader甚至ResourcePatternResolver。而这就是ApplicationContext支持Spring内统一资源加载策略的真相。

通常，所有的ApplicationContext实现类会直接或者间接地继承org.springframework.context.support.AbstractApplicationContext，从这个类上，我们就可以看到ApplicationContext与ResourceLoader之间的所有关系。AbstractApplicationContext继承了DefaultResourceLoader，那么，它的getResource(String)当然就直接用DefaultResourceLoader的了。剩下需要它“效劳”的，就是ResourcePatternResolver的Resource[]getResources (String)，当然，AbstractApplicationContext也不负众望，当即拿下。AbstractApplicationContext类的内部声明有一个resourcePatternResolver，类型是ResourcePatternResolver，对应的实例类型为PathMatchingResourcePatternResolver。之前我们说过PathMatchingResourcePatternResolver构造的时候会接受一个ResourceLoader，而AbstractApplicationContext本身又继承自DefaultResourceLoader，当然就直接把自身给“贡献”了。这样，整个ApplicationContext的实现类就完全可以支持ResourceLoader或者ResourcePatternResolver接口，你能说ApplicationContext不支持Spring的统一资源加载吗？说白了，ApplicationContext的实现类在作为ResourceLoader或者ResourcePatternResolver时候的行为，完全就是委派给了PathMatchingResourcePatternResolver和DefaultResourceLoader来做。图5-2给出了AbstractApplicationContext与ResourceLoader和ResourcePatternResolver之间的类层次关系。

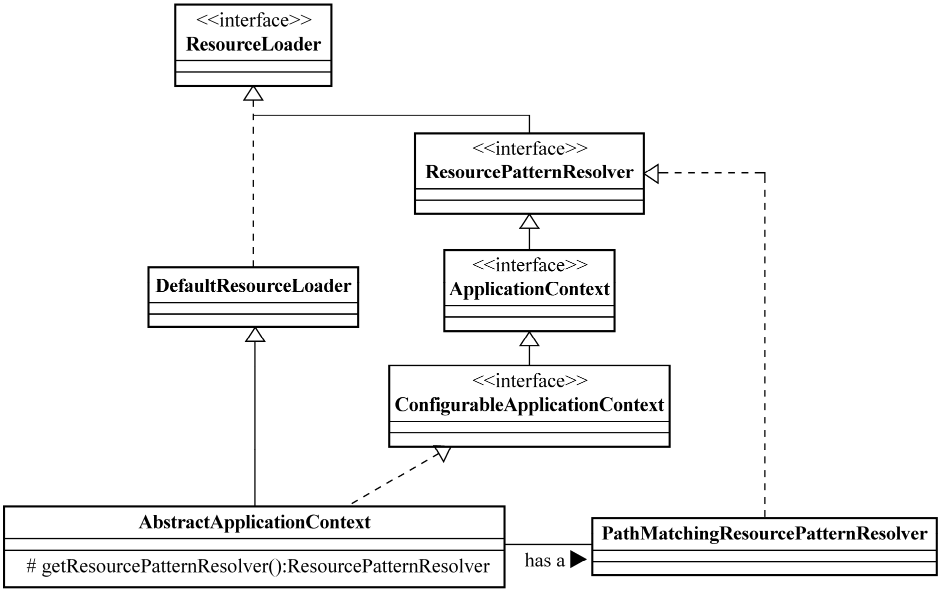

图5-2　AbstractApplicationContext作为ResourceLoader和ResourcePatternResolver

有了这些做前提，让我们看看作为ResourceLoader或者ResourcePatternResolver的ApplicationContext，到底因此拥有了何等神通吧！

1. 扮演ResourceLoader的角色

既然ApplicationContext可以作为ResourceLoader或者ResourcePatternResolver来使用，那么，很显然，我们可以通过ApplicationContext来加载任何Spring支持的Resource类型。与直接使用ResourceLoader来做这些事情相比，很明显，ApplicationContext的表现过于“谦虚”了。代码清单5-5演示的正是“大材小用”后的ApplicationContext。

代码清单5-5　以ResourceLoader身份登场的ApplicationContext

```
ResourceLoader resourceLoader = new ClassPathXmlApplicationContext("配置文件路径");
// 或者
// ResourceLoader resourceLoader = new FileSystemXmlApplicationContext("配置文件路径");

Resource fileResource = resourceLoader.getResource("D:/spring21site/README");
assertTrue(fileResource instanceof ClassPathResource);
assertFalse(fileResource.exists());

Resource urlResource2 = resourceLoader.getResource("http://www.spring21.cn");
assertTrue(urlResource2 instanceof UrlResource);
```

我想这样的使用场景，你一定比我先猜到，不是吗？

2. ResourceLoader类型的注入

在大部分情况下，如果某个bean需要依赖于ResourceLoader来查找定位资源，我们可以为其注入容器中声明的某个具体的ResourceLoader实现，该bean也无需实现任何接口，直接通过构造方法注入或者setter方法注入规则声明依赖即可，这样处理是比较合理的。不过，如果你不介意你的bean定义依赖于Spring的API，那不妨考虑用一下Spring提供的便利。

4.4.3节中曾经提到几个对ApplicationContext特定的Aware接口，这其中就包括Resource-LoaderAware和ApplicationContextAware接口。

假设我们有类定义如代码清单5-6所示。

代码清单5-6 依赖于ResourceLoader的实例类

```
public class FooBar {
  private ResourceLoader resourceLoader;

  public void foo(String location)
  {
    System.out.println(getResourceLoader().getResource(location).getClass());
  }

  public ResourceLoader getResourceLoader() {
    return resourceLoader;
  }
  public void setResourceLoader(ResourceLoader resourceLoader) {
    this.resourceLoader = resourceLoader;
  }
}
```

该类出于什么目的要依赖于ResourceLoader，我们暂且不论，要为其注入什么样的Resource-Loader实例才是我们当下该操心的事情。姑且先给它注入DefaultResourceLoader。这样也就有了如下配置：

```
<bean id="resourceLoader" class="org.springframework.core.io.DefaultResourceLoader">
</bean>

<bean id="fooBar" class="...FooBar">
  <property name="resourceLoader">
    <ref bean="resourceLoader"/>
  </property>
</bean>
```

不过，ApplicationContext容器本身就是一个ResourceLoader，我们为了该类还需要单独提供一个resourceLoader实例就有些多于了，直接将当前的ApplicationContext容器作为Resource-Loader注入不就行了？而ResourceLoaderAware和ApplicationContextAware接口正好可以帮助我们做到这一点，只不过现在的FooBar需要依赖于Spring的API了。不过，在我看来，这没有什么大不了，因为我们从来也没有真正逃脱过依赖（这种依赖也好，那种依赖也罢）。

现在，修改我们的FooBar定义，让其实现ResourceLoaderAware或者Application-ContextAware接口，修改后的定义如代码清单5-7所示。

代码清单5-7 实现了ResourceLoaderAware或者ApplicationContextAware接口的实例类

```
public class FooBar implements ResourceLoaderAware{
  private ResourceLoader resourceLoader;

  public void foo(String location)
  {
  System.out.println(getResourceLoader().getResource(location).getClass());
  }

  public ResourceLoader getResourceLoader() {
    return resourceLoader;
  }
```

```
    public void setResourceLoader(ResourceLoader resourceLoader) {
      this.resourceLoader = resourceLoader;
    }
  }
  public class FooBar implements ApplicationContextAware{
    private ResourceLoader resourceLoader;

    public void foo(String location)
    {
    System.out.println(getResourceLoader().getResource(location).getClass());
    }

    public ResourceLoader getResourceLoader() {
      return resourceLoader;
    }
    public void setApplicationContext(ApplicationContext ctx) ➥
    throws BeansException {
    this.resourceLoader = ctx;
    }
  }
```

剩下的就是直接将一个FooBar配置到bean定义文件即可，如下所示：

```
<bean id="fooBar" class="...FooBar">
</bean>
```

哇，简洁多了不是嘛？现在，容器启动的时候，就会自动将当前ApplicationContext容器本身注入到FooBar中，因为ApplicationContext类型容器可以自动识别Aware接口。

当然，如果应用场景仅使用ResourceLoader类型即可满足需求，那么，还是使用ResourceLoaderAware比较合适，ApplicationContextAware相对来说过于宽泛了些（当然，使用也未尝不可）。

3. Resource类型的注入

我们之前讲过，容器可以将bean定义文件中的字符串形式表达的信息，正确地转换成具体对象定义的依赖类型。对于那些Spring容器提供的默认的PropertyEditors无法识别的对象类型，我们可以提供自定义的PropertyEditor实现并注册到容器中，以供容器做类型转换的时候使用。默认情况下，BeanFactory容器不会为org.springframework.core.io.Resource类型提供相应的PropertyEditor，所以，如果我们想注入Resource类型的bean定义，就需要注册自定义的PropertyEditor到BeanFactory容器。不过，对于ApplicationContext来说，我们无需这么做，因为ApplicationContext容器可以正确识别Resource类型并转换后注入相关对象。

假设有一个XMailer类，它依赖于一个模板来提供邮件发送的内容，我们声明模板为Resource类型，那么，最终的XMailer定义也就如代码清单5-8所示。

代码清单5-8　依赖于Resource的XMailer类定义

```
public class XMailer {
  private Resource template;

  public void sendMail(Map mailCtx)
  {
    // String mailContext = merge(getTemplate().getInputStream(),mailCtx);
    //...
  }

  public Resource getTemplate() {
    return template;
```

```
    }

    public void setTemplate(Resource template) {
      this.template = template;
    }
}
```

该类定义与平常的bean定义没有什么差别，我们直接在配置文件中以String形式指定template所在位置，ApplicatonContext就可以正确地转换类型并注入依赖，配置内容如下：

```
<bean id="mailer" class="...XMailer">
  <property name="template" value="..resources.default_template.vm"/>
  ...
</bean>
```

至于这里面的奥秘，估计你也猜个八九不离十了。

ApplicationContext启动伊始，会通过一个org.springframework.beans.support.ResourceEditorRegistrar来注册Spring提供的针对Resource类型的PropertyEditor实现到容器中，这个PropertyEditor叫做org.springframework.core.io.ResourceEditor。这样，ApplicationContext就可以正确地识别Resource类型的依赖了。至于ResourceEditor怎么实现我就不用说了吧？你想啊，把配置文件中的路径让ApplicationContext作为ResourceLoader给你定位一下不就得了。

注意 如果应用对象需要依赖一组Resource，与ApplicationContext注册了ResourceEditor类似，Spring提供了org.springframework.core.io.support.ResourceArrayPropertyEditor实现，我们只需要通过CustomEditorConfigurar告知容器即可。

4. 在特定情况下，ApplicationContext的Resource加载行为

特定的ApplicationContext容器实现，在作为ResourceLoader加载资源时，会有其特定的行为。我们下面主要讨论两种类型的ApplicationContext容器，即ClassPathXmlApplicationContext和FileSystemXmlApplicationContext。其他类型的ApplicationContext容器，会在稍后章节中提到。

我们知道，对于URL所接受的资源路径来说，通常开始都会有一个协议前缀，比如file:、http:、ftp:等。既然Spring使用UrlResource对URL定位查找的资源进行了抽象，那么，同样也支持这样类型的资源路径，而且，在这个基础上，Spring还扩展了协议前缀的集合。ResourceLoader中增加了一种新的资源路径协议——classpath:，ResourcePatternResolver又增加了一种——classpath*:。这样，我们就可以通过这些资源路径协议前缀，明确地告知Spring容器要从classpath中加载资源，如下所示：

```
// 代码中使用协议前缀
ResourceLoader resourceLoader = new ➥
FileSystemXmlApplicationContext("classpath:conf/container-conf.xml");
// 配置中使用协议前缀
<bean id="..." class="...">
  <property name="...">
    <value>classpath:resource/template.vm</value>
  </property>
</bean>
```

classpath*:与classpath:的唯一区别就在于，如果能够在classpath中找到多个指定的资源，则返回多个。我们可以通过这两个前缀改变某些ApplicationContext实现类的默认资源加载行为。

ClassPathXmlApplicationContext和FileSystemXmlApplicationContext在处理资源加载的

默认行为上有所不同。当ClassPathXmlApplicationContext在实例化的时候，即使没有指明classpath:或者classpath*:等前缀，它会默认从classpath中加载bean定义配置文件，以下代码中演示的两种实例化方式效果是相同的：

```
ApplicationContext ctx = new ClassPathXmlApplicationContext("conf/appContext.xml");
以及
ApplicationContext ctx = new ClassPathXmlApplicationContext("classpath:conf/appContext.xml");
```

而FileSystemXmlApplicationContext则有些不同，如果我们像如下代码那样指定conf/appContext.xml，它会尝试从文件系统中加载bean定义文件：

```
ApplicationContext ctx = new FileSystemXmlApplicationContext("conf/appContext.xml");
```

不过，我们可以像如下代码所示，通过在资源路径之前增加classpath:前缀，明确指定FileSystemXmlApplicationContext从classpath中加载bean定义的配置文件：

```
ApplicationContext ctx = new FileSystemXmlApplicationContext("classpath:conf/appContext.xml");
```

这时，FileSystemXmlApplicationContext就是从Classpath中加载配置，而不是从文件系统中加载。也就是说，它现在对应的是ClassPathResource类型的资源，而不是默认的FileSystem-Resource类型资源。FileSystemXmlApplicationContext之所以如此，是因为它与org.springfra-mework.core.io.FileSystemResourceLoader一样，也覆写了DefaultResourceLoader的getRes-ourceByPath(String)方法，逻辑跟 FileSystemResourceLoader一模一样。

当实例化相应的ApplicationContext时，各种实现会根据自身的特性，从不同的位置加载bean定义配置文件。当容器实例化并启动完毕，我们要用相应容器作为ResourceLoader来加载其他资源时，各种ApplicationContext容器的实现类依然会有不同的表现。

对于ClassPathXmlApplicationContext来说，如果我们不指定路径之前的前缀，它也不会像资源路径所表现的那样，从文件系统加载资源，而是像实例化时候的行为一样，从Classpath中加载这种没有路径前缀的资源。如类似如下指定的资源路径，ClassPathXmlApplicationContext依然尝试从Classpath加载：

```
<bean id="..." class="...">
  <property name="..." value="conf/appContext.xml"/>
</bean>
```

如果当前容器类型为FileSystemXmlApplicationContext，事情则会像预想的那样进行，FileSystemXmlApplicationContext将从文件系统中给我们加载该文件。但是，就跟实例化时可以通过classpath:前缀覆盖掉FileSystemXmlApplicationContext的默认加载行为一样，我们也可以在这个时候用classpath:前缀强制指定FileSystemXmlApplicationContext从Classpath中加载该文件，如以下代码所示：

```
<bean id="..." class="...">
  <property name="..." value="classpath:conf/appContext.xml"/>
</bean>
```

去掉配置中的classpath:前缀，FileSystemXmlApplicationContext默认从文件系统加载资源。

小心 即使在FileSystemXmlApplicationContext实例化启动时，通过classpath:前缀强制让它从Classpath中加载bean定义文件，但这也仅限于容器的实例化并加载bean定义文件这个特

定阶段。容器实例化并启动后，作为ResourceLoader来加载资源，如果不是每个地方都使用classpath:前缀，强制FileSystemXmlApplicationContext从Classpath中加载资源，FileSystemXmlApplicationContext还会默认从文件系统中加载资源。

如果细化下去，这部分内容还有许多，如通配符加载的行为、FileSystemResource的特定行为等。这里不做赘述，更多相关特性，请参照Spring参考文档。

5.2　国际化信息支持（I18n[①] MessageSource）

全世界有很多不同的国家和地区，每个国家或者地区都使用各自的语言文字。在当今全球化的信息大潮中，要让我们的应用程序可以供全世界不同国家和地区的人们使用，应用程序就必须支持它所面向的国家和地区的语言文字，为不同的国家和地区的用户提供他们各自的语言文字信息。所以，要向全世界推广，应用程序的国际化信息支持自然是势在必行。

5.2.1　Java SE 提供的国际化支持

程序的国际化不是三言两语可以讲清楚的，它涉及许多的内容，如货币形式的格式化、时间的表现形式、各国家和地区的语言文字等。要全面了解Java中的I18n，建议参考O'Reilly出版的*Java Internationalization*。我们这里主要是简单地介绍基本概念，以便你可以对Spring中的国际化信息支持有更好的了解。

对于Java中的国际化信息处理，主要涉及两个类，即java.util.Locale和java.util.ResourceBundle。

1. **Locale**

不同的Locale代表不同的国家和地区，每个国家和地区在Locale这里都有相应的简写代码表示，包括语言代码以及国家代码，这些代码是ISO标准代码。如，Locale.CHINA代表中国，它的代码表示为zh_CN；Locale.US代表美国地区，代码表示为en_US；而美国和英国等都属于英语地区，则可以使用Locale.ENGLISH来统一表示，这时代码只有语言代码，即en。

Locale类提供了三个构造方法，它们的定义如下：

```
Locale(String language)
Locale(String language, String country)
Locale(String language, String country, String variant)
```

这样我们就可以根据相应的语言代码和国家代码来构造相应的Locale，如以下所示：

```
Locale china = new Locale("zh","CN");
相当于
Locale.CHINA
```

常用的Locale都提供有静态常量，不用我们自己重新构造。一些不常用的Locale的则需要根据相应的国家和地区以及语言来进行构造。有了Locale，我们的应用程序就可以通过它来判别如何为不同的国家和地区的用户提供相应的信息。

2. **ResourceBundle**

ResourceBundle用来保存特定于某个Locale的信息（可以是String类型信息，也可以是任何类型的对象）。通常，ResourceBundle管理一组信息序列，所有的信息序列有统一的一个basename，然后特定的Locale的信息，可以根据basename后追加的语言或者地区代码来区分。比如，我们用一组

① Internationalization，单词的开头字母I和结尾字母n中间有18个字母，所以，通常简写为I18n。

properties文件来分别保存不同国家地区的信息，可以像下面这样来命名相应的properties文件：

```
messages.properties
messages_zh.properties
messages_zh_CN.properties
messages_en.properties
messages_en_US.properties
...
```

其中，文件名中的messages部分称作ResourceBundle将加载的资源的basename，其他语言或地区的资源在basename的基础上追加Locale特定代码。

每个资源文件中都有相同的键来标志具体资源条目，但每个资源内部对应相同键的资源条目内容，则根据Locale的不同而不同。如下代码片段演示了两个不同的资源文件内容的对比情况：

```
# messages_zh_CN.properties文件中
menu.file=文件({0})
menu.edit=编辑
...

# messages_en_US.properties文件中
menu.file=File({0})
menu.edit=Edit
...
```

注意　按照规定，properties文件内容是以ISO-8859-1编码的，所以，实际上message_zh_CN.properties中各个键对应的内容是不应该以中文提供的，应该使用native2ascii或者类似的相关工具进行转码，这里如此举例，只是为了更好地说明差别。

有了ResourceBundle对应的资源文件之后，我们就可以通过ResourceBundle的getBundle(String baseName, Locale locale)方法取得不同Locale对应的ResourceBundle，然后根据资源的键取得相应Locale的资源条目内容。

通过结合ResourceBundle和Locale，我们就能够实现应用程序的国际化信息支持。

5.2.2　MessageSource与ApplicationContext

Spring在Java SE的国际化支持的基础上，进一步抽象了国际化信息的访问接口，也就是org.springframework.context.MessageSource，该接口定义如下：

```
public interface MessageSource {

   String getMessage(String code, Object[] args, String defaultMessage, Locale locale);

   String getMessage(String code, Object[] args, Locale locale) throws NoSuchMessageException;

   String getMessage(MessageSourceResolvable resolvable, Locale locale) throws ➥
   NoSuchMessage zException;

}
```

通过该接口，我们统一了国际化信息的访问方式。传入相应的Locale、资源的键以及相应参数，就可以取得相应的信息，再也不用先根据Locale取得ResourceBundle，然后再从ResourceBundle查询信息了。对MessageSource所提供的三个方法的简单说明如下。

- String getMessage(String code, Object[] args, String defaultMessage, Locale locale)。根据传入的资源条目的键（对应方法声明中的code参数）、信息参数以及Locale来

查找信息，如果对应信息没有找到，则返回指定的defaultMessage。

- String getMessage(String code, Object[] args, Locale locale) throws NoSuchMessageException。与第一个方法相同，只不过，因为没有指定默认信息，当对应的信息找不到的情况下，将抛出NoSuchMessageException异常。
- String getMessage(MessageSourceResolvable resolvable, Locale locale) throws NoSuchMessageException。使用MessageSourceResolvable对象对资源条目的键、信息参数等进行封装，将封住了这些信息的MessageSourceResolvable对象作为查询参数来调用以上方法。如果根据MessageSourceResolvable中的信息查找不到相应条目内容，将抛出NoSuchMessageException异常。

现在我们知道，ApplicationContext除了实现了ResourceLoader以支持统一的资源加载，它还实现了MessageSource接口，那么就跟ApplicationContext因为实现了ResourceLoader而可以当作ResourceLoader来使用一样，ApplicationContext现在也是一个MessageSource了。

在默认情况下，ApplicationContext将委派容器中一个名称为messageSource的MessageSource接口实现来完成MessageSource应该完成的职责。如果找不到这样一个名字的MessageSource实现，ApplicationContext内部会默认实例化一个不含任何内容的StaticMessageSource实例，以保证相应的方法调用。所以通常情况下，如果要提供容器内的国际化信息支持，我们会添加如代码清单5-9类似的配置信息到容器的配置文件中。

代码清单5-9　ApplicationContext容器内使用的messageSource的配置实例

```
<beans>
  <bean id="messageSource" class="org.springframework.context.support. ➥
  ResourceBundleMessageSource">
    <property name="basenames">
      <list>
        <value>messages</value>
        <value>errorcodes</value>
      </list>
    </property>
  </bean>
  ...
</beans>
```

有了这些，我们就可以通过ApplicationContext直接访问相应Locale对应的信息，如下所示：

```
ApplicationContext ctx = ...;
String fileMenuName = ctx.getMessage("menu.file", new Object[]{"F"}, Locale.US);
String editMenuName = ctx.getMessage("menu.file", null, Locale.US);
assertEquals("File(F)", fileMenuName);
assertEquals("Edit", editMenuName);
```

1. 可用的MessageSource实现

Spring提供了三种MessageSource的实现，即StaticMessageSource、ResourceBundleMessageSource和ReloadableResourceBundleMessageSource。

- org.springframework.context.support.StaticMessageSource。MessageSource接口的简单实现，可以通过编程的方式添加信息条目，多用于测试，不应该用于正式的生产环境。
- org.springframework.context.support.ResourceBundleMessageSource。基于标准的java.util.ResourceBundle而实现的MessageSource，对其父类AbstractMessageSource的行为进行了扩展，提供对多个ResourceBundle的缓存以提高查询速度。同时，对于参数化的信息和非参数化信息的处理进行了优化，并对用于参数化信息格式化的MessageFormat实

例也进行了缓存。它是最常用的、用于正式生产环境下的MessageSource实现。

- org.springframework.context.support.ReloadableResourceBundleMessageSource。同样基于标准的java.util.ResourceBundle而构建的MessageSource实现类，但通过其cacheSeconds属性可以指定时间段，以定期刷新并检查底层的properties资源文件是否有变更。对于properties资源文件的加载方式也与ResourceBundleMessageSource有所不同，可以通过ResourceLoader来加载信息资源文件。使用ReloadableResourceBundleMessageSource时，应该避免将信息资源文件放到classpath中，因为这无助于ReloadableResourceBundleMessageSource定期加载文件变更。更多信息参照该类的Javadoc。

这三种实现都可以独立于容器并在独立运行（Standalone形式）的应用程序中使用，而并非只能依托ApplicationContext才可使用。代码清单5-10为我们演示了这三种MessageSource的简单使用。

代码清单5-10　三种MessageSource实现类的简单使用演示

```
StaticMessageSource messageSource = new StaticMessageSource();
messageSource.addMessage("menu.file", Locale.US, "File");
messageSource.addMessage("menu.edit", Locale.US, "Edit");
assertEquals("File(F)", messageSource.getMessage("menu.file", new Object[]{"F"}, Locale.US));
assertEquals("Edit", messageSource.getMessage("menu.edit", null,"Edit", Locale.US));

ResourceBundleMessageSource messageSource = new ResourceBundleMessageSource();
messageSource.setBasenames(new String[]{"conf/messages"});// 从 classpath加载资源文件
assertEquals("File(F)", messageSource.getMessage("menu.file", new Object[]{"F"}, Locale.US));

ReloadableResourceBundleMessageSource messageSource = new ➥
ReloadableResourceBundleMessageSource();
messageSource.setBasenames(new String[]{"file:conf/messages"}); // 从文件系统加载资源文件
assertEquals("File(F)", messageSource.getMessage("menu.file", new Object[]{"F"},➥
Locale.US));
```

之前提到，ApplicationContext需要其配置文件中有一个名称为messageSource的MessageSource实现，自然就是以上的三选一了。

至此，我们有了图5-3。

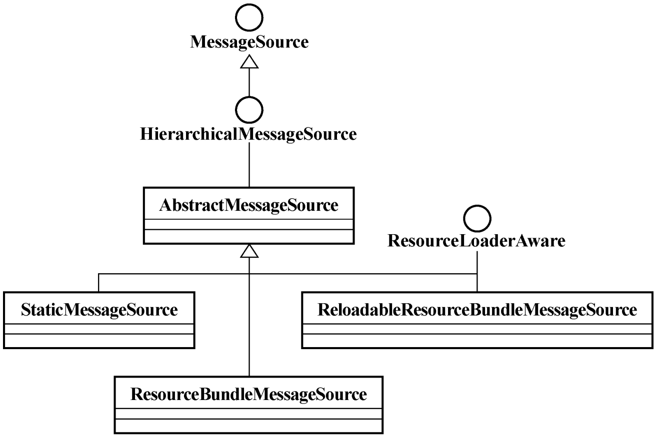

图5-3　MessageSource类层次结构

如果说以上三种MessageSource的实现还是不能满足你的要求，那么直接继承Abstract-

MessageSource，然后覆写几个方法就行了，甚至可以直接实现MessageSource接口，如果你的需求真的那么特别。

2. **MessageSourceAware和MessageSource的注入**

ApplicationContext启动的时候，会自动识别容器中类型为MessageSourceAware的bean定义，并将自身作为MessageSource注入相应对象实例中。如果某个业务对象需要国际化的信息支持，那么最简单的办法就是让它实现MessageSourceAware接口，然后注册到ApplicationContext容器。不过这样一来，该业务对象对ApplicationContext容器的依赖性就太强了，显得容器具有较强的侵入性。

而实际上，如果真的某个业务对象需要依赖于MessageSource的话，直接通过构造方法注入或者setter方法注入的方式声明依赖就可以了。只要配置bean定义时，将ApplicationContext容器内部的那个messageSource注入该业务对象即可。假设我们有一个通用的Validator数据验证类，它需要通过MessageSource来返回相应的错误信息，那么可以为其声明一个MessageSource依赖，然后将ApplicationContext中的那个已经配置好的messageSource注入给它。代码清单5-11给出了该类的定义以及相关注入配置。

代码清单5-11　依赖于MessageSource的Validator类定义以及相关注入配置

```
public class Validator
{
  private MessageSource messageSource;

  public ValidateResult validate(Object target)
  {
    // 执行相应验证逻辑
    // 如果有错误，通过messageSource.getMessage(...)获取相应信息并放入验证结果对象中
    // 返回验证结果（return result）
  }
  public MessageSource getMessageSource()
  {
    return messageSource;
  }
  public void setMessageSource(MessageSource msgSource)
  {
    this.messageSource = msgSource;
  }
  // ...
}

<beans>
<bean id="messageSource" ➥
class="org.springframework.context.support.ResourceBundleMessageSource">
  <property name="basenames">
    <list>
      <value>messages</value>
      <value>errorcodes</value>
    </list>
  </property>
</bean>

<bean id="validator" class="...Validator">
  <property name="messageSource" ref="messageSource"/>
</bean>
...
</beans>
```

与通常的依赖注入没有什么区别，不是吗？

既然MessageSource可以独立使用，那为什么还让ApplicationContext实现该接口呢？在独立运行的应用程序（Standalone Application）中，就如我们上面这些应用场景所展示的那样，直接使用MessageSource的相应实现类就行了。不过在Web应用程序中，通常会公开ApplicationContext给视图（View）层，这样，通过标签（tag）就可以直接访问国际化信息了。我们将在第六部分更多接触相关内容，至此，ApplicationContext对国际化信息的支持功能的讲解告一段落。

5.3　容器内部事件发布

Spring的ApplicationContext容器提供的容器内事件发布功能，是通过提供一套基于Java SE标准自定义事件类而实现的。为了更好地了解这组自定义事件类，我们可以先从Java SE的标准自定义事件类实现的推荐流程说起。

5.3.1　自定义事件发布

Java SE提供了实现自定义事件发布（Custom Event publication）功能的基础类，即java.util.EventObject类和java.util.EventListener接口。所有的自定义事件类型可以通过扩展EventObject来实现，而事件的监听器则扩展自EventListener。下面让我们看一下要实现一套自定义事件发布类的架构，应该如何来做。

给出自定义事件类型（define your own event object）。为了针对具体场景可以区分具体的事件类型，我们需要给出自己的事件类型的定义，通常做法是扩展java.util.EventObject类来实现自定义的事件类型。我们此次定义的自定义事件类型见代码清单5-12。

代码清单5-12　针对方法执行事件的自定义事件类型定义

```
public class MethodExecutionEvent extends EventObject {
   private static final long serialVersionUID = -71960369269303337L;
   private String methodName;

   public MethodExecutionEvent(Object source) {
      super(source);
   }
   public MethodExecutionEvent(Object source,String methodName)
   {
      super(source);
      this.methodName = methodName;
   }
   public String getMethodName() {
      return methodName;
   }
   public void setMethodName(String methodName) {
      this.methodName = methodName;
   }
}
```

我们想对方法的执行情况进行发布和监听，所以，就声明了一个MethodExecutionEvent类型，它继承自EventObject，当该类型的事件发布之后，相应的监听器即可对该类型的事件进行处理。如果需要，自定义事件类可以根据情况提供更多信息，不用担心自定义事件类的“承受力”。

实现针对自定义事件类的事件监听器接口（define custom event listener）。自定义的事件监听器需要在合适的时机监听自定义的事件，如刚声明的MethodExecutionEvent，我们可以在方法开始执行的时候发布该事件，也可以在方法执行即将结束之际发布该事件。相应地，自定义的事件监听器

需要提供方法对这两种情况下接收到的事件进行处理。代码清单5-13给出了针对MethodExecutionEvent的事件监听器接口定义。

代码清单5-13 自定义事件监听器MethodExecutionEventListener定义

```
public interface MethodExecutionEventListener extends EventListener {
   /**
    * 处理方法开始执行的时候发布的MethodExecutionEvent事件
    */
   void onMethodBegin(MethodExecutionEvent evt);
   /**
    * 处理方法执行将结束时候发布的MethodExecutionEvent事件
    */
   void onMethodEnd(MethodExecutionEvent evt);
}
```

事件监听器接口定义首先继承了java.util.EventListener,然后针对不同的事件发布时机提供相应的处理方法定义，最主要的就是，这些处理方法所接受的参数就是MethodExecutionEvent类型的事件。也就是说，我们的自定义事件监听器类只负责监听其对应的自定义事件并进行处理，如果什么事件它都要处理，那么非忙死不可。有了事件监听器接口定义，还必须根据时机需求提供相应的实现，只有接口定义可是干不了什么事情的啊！出于简化，我们仅给出一个简单的实现定义，见代码清单5-14。

代码清单5-14 自定义事件监听器具体实现类SimpleMethodExecutionEventListener的定义

```
public class SimpleMethodExecutionEventListener implements ➥
MethodExecutionEventListener {

   public void onMethodBegin(MethodExecutionEvent evt) {
      String methodName = evt.getMethodName();
      System.out.println("start to execute the method["+methodName+"].");
   }

   public void onMethodEnd(MethodExecutionEvent evt) {
      String methodName = evt.getMethodName();
      System.out.println("finished to execute the method["+methodName+"].");
   }
}
```

组合事件类和监听器，发布事件。有了自定义事件和自定义事件监听器，剩下的就是发布事件，然后让相应的监听器监听并处理事件了。通常情况下，我们会有一个事件发布者（EventPublisher），它本身作为事件源，会在合适的时点，将相应事件发布给对应的事件监听器。代码清单5-15给出了针对MethodExecutionEvent的事件发布者类的定义。

代码清单5-15 MethodExeuctionEventPublisher时间发布者类定义

```
public class MethodExeuctionEventPublisher {

   private List<MethodExecutionEventListener> listeners = new ➥
   ArrayList<MethodExecutionEventListener>();

   public void methodToMonitor()
   {
      MethodExecutionEvent event2Publish = ➥
      new MethodExecutionEvent(this,"methodToMonitor");
      publishEvent(MethodExecutionStatus.BEGIN,event2Publish);
```

```
        // 执行实际的方法逻辑
        // ...
        publishEvent(MethodExecutionStatus.END,event2Publish);
    }

    protected void publishEvent(MethodExecutionStatus status, ➥
    MethodExecutionEvent methodExecutionEvent) {
        List<MethodExecutionEventListener> copyListeners = ➥
        new ArrayList<MethodExecutionEventListener>(listeners);
        for(MethodExecutionEventListener listener:copyListeners)
        {
            if(MethodExecutionStatus.BEGIN.equals(status))
                listener.onMethodBegin(methodExecutionEvent);
            else
                listener.onMethodEnd(methodExecutionEvent);
        }
    }

    public void addMethodExecutionEventListener(MethodExecutionEventListener listener)
    {
        this.listeners.add(listener);
    }
    public void removeListener(MethodExecutionEventListener listener)
    {
        if(this.listeners.contains(listener))
            this.listeners.remove(listener);
    }
    public void removeAllListeners()
    {
        this.listeners.clear();
    }

    public static void main(String[] args) {
        MethodExeuctionEventPublisher eventPublisher = ➥
        new MethodExeuctionEventPublisher();
        eventPublisher.addMethodExecutionEventListener(new ➥
        SimpleMethodExecutionEventListener());
        eventPublisher.methodToMonitor();
    }
}
```

我们的事件发布者关注的主要有两点。

具体时点上自定义事件的发布。方法methodToMonitor()是事件发布的源头，MethodExeuctionEventPublisher在该方法开始和即将结束的时候，分别针对这两个时点发布MethodExecutionEvent事件。具体实现上，每个时点发布的事件会通过MethodExecutionEventListener的相应方法传给注册的监听者并被处理掉。在实现中，需要注意到，为了避免事件处理期间事件监听器的注册或移除操作影响处理过程，我们对事件发布时点的监听器列表进行了一个安全复制（safe-copy）。另外，事件的发布是顺序执行，所以为了能够不影响处理性能，事件监听器的处理逻辑应该尽量简短。

自定义事件监听器的管理。MethodExeuctionEventPublisher类提供了与事件监听器的注册和移除相关的方法，这样，客户端可以根据情况决定是否需要注册或者移除某个事件监听器。这里容易出现问题的情况是，如果没有提供remove事件监听器的方法，那么注册的监听器实例会一直被MethodExeuctionEventPublisher引用，即使已经过期了或者废弃不用了，也依然存在于MethodExeuctionEventPublisher的监听器列表中。这会导致隐性的内存泄漏，在任何事件监听器的处理上都可能出现这种问题。

整个Java SE中标准的自定义事件实现就是这个样子，基本上涉及三个角色，即自定义的事件类型、自定义的事件监听器和自定义的事件发布者，关系如图5-4所示。

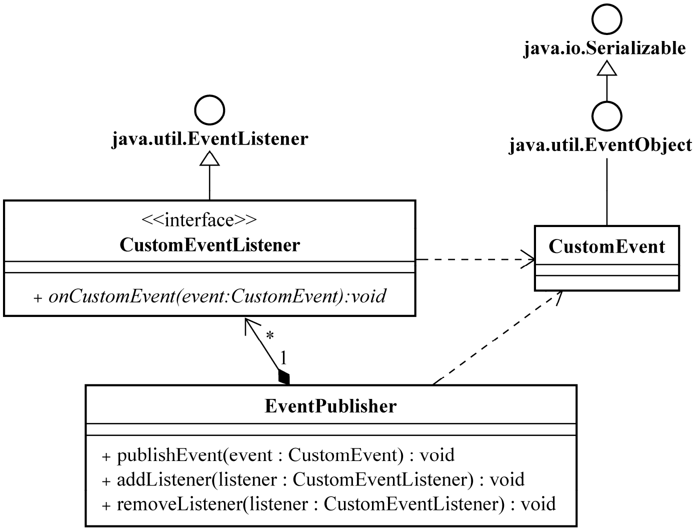

图5-4 自定义事件发布类结构图

5.3.2 Spring 的容器内事件发布类结构分析

Spring 的 ApplicationContext 容器内部允许以 org.springframework.context.ApplicationEvent 的形式发布事件，容器内注册的 org.springframework.context.ApplicationListener类型的bean定义会被ApplicationContext容器自动识别，它们负责监听容器内发布的所有ApplicationEvent类型的事件。也就是说，一旦容器内发布ApplicationEvent及其子类型的事件，注册到容器的ApplicationListener就会对这些事件进行处理。

我想你已经猜到是怎么回事了。

- ApplicationEvent

Spring容器内自定义事件类型，继承自java.util.EventObject，它是一个抽象类，需要根据情况提供相应子类以区分不同情况。默认情况下，Spring提供了三个实现。

- ContextClosedEvent：ApplicationContext容器在即将关闭的时候发布的事件类型。
- ContextRefreshedEvent：ApplicationContext容器在初始化或者刷新的时候发布的事件类型。
- RequestHandledEvent：Web请求处理后发布的事件，其有一子类ServletRequestHandledEvent提供特定于Java EE的Servlet相关事件。

- ApplicationListener

ApplicationContext容器内使用的自定义事件监听器接口定义，继承自java.util.EventListener。ApplicationContext容器在启动时，会自动识别并加载EventListener类型bean定义，一旦容器内有事件发布，将通知这些注册到容器的EventListener。

- ApplicationContext

还记得ApplicationContext的定义吧？除了之前的ResourceLoader和MessageSource，ApplicationContext接口定义还继承了ApplicationEventPublisher接口，该接口提供了void publishEvent(ApplicationEvent event)方法定义。不难看出，ApplicationContext容器现在担当的就是事件发布者的角色。

虽然ApplicationContext继承了ApplicationEventPublisher接口而担当了事件发布者的角色，但是在具体实现上，与之前提到的自定义事件实现流程有些许差异，且让我一一道来……

ApplicationContext容器的具体实现类在实现事件的发布和事件监听器的注册方面，并没事必躬亲，而是把这些活儿转包给了一个称作org.springframework.context.event.ApplicationEventMulticaster的接口。该接口定义了具体事件监听器的注册管理以及事件发布的方法，但接口终归是接口，还得有具体实现。ApplicationEventMulticaster有一抽象实现类——org.springframework.context.event.AbstractApplicationEventMulticaster，它实现了事件监听器的管理功能。出于灵活性和扩展性考虑，事件的发布功能则委托给了其子类。org.springframework.context.event.SimpleApplicationEventMulticaster是Spring提供的AbstractApplicationEventMulticaster的一个子类实现，添加了事件发布功能的实现。不过，其默认使用了SyncTaskExecutor进行事件的发布。与我们给出的样例事件发布者实现一样，事件是同步顺序发布的。为了避免这种方式可能存在的性能问题，我们可以为其提供其他类型的TaskExecutor实现类（TaskExecutor的概念将在后面详细介绍）。

因为ApplicationContext容器的事件发布功能全部委托给了ApplicationEventMulticaster来做，所以，容器启动伊始，就会检查容器内是否存在名称为applicationEventMulticaster的ApplicationEventMulticaster对象实例。有的话就使用提供的实现，没有则默认初始化一个SimpleApplicationEventMulticaster作为将会使用的ApplicationEventMulticaster。这样，整个Spring容器内事件发布功能实现结构图就有了，如图5-5所示。

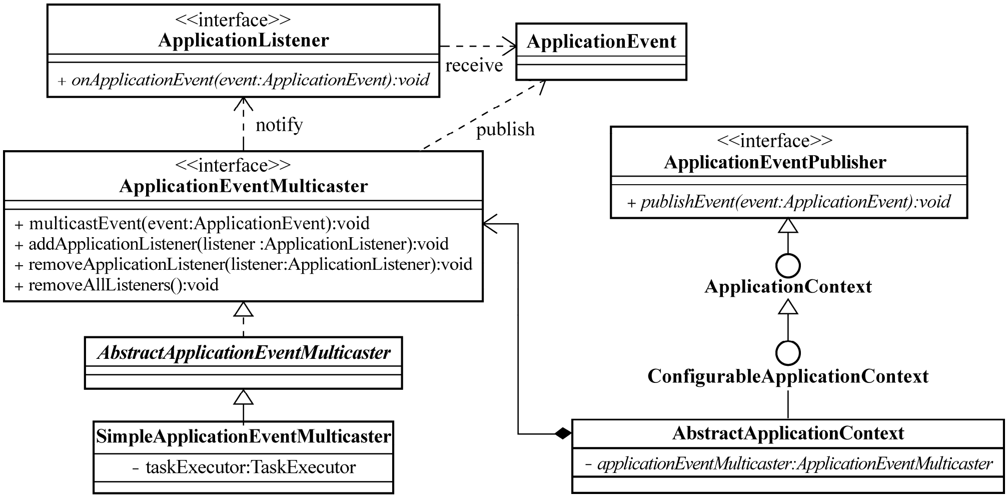

图5-5　Spring容器内事件发布实现类图

5.3.3　Spring 容器内事件发布的应用

Spring的ApplicationContext容器内的事件发布机制，主要用于单一容器内的简单消息通知和处理，并不适合分布式、多进程、多容器之间的事件通知。虽然可以通过Spring的Remoting支持，“曲折一点”来实现较为复杂的需求，但是难免弊大于利，失大于得。其他消息机制处理较复杂场景或许更合适。所以，我们应该在合适的地点、合适的需求分析的前提下，合理地使用Spring提供的ApplicationContext容器内的事件发布机制。

要让我们的业务类支持容器内的事件发布，需要它拥有ApplicationEventPublisher的事件发布支持。所以，需要为其注入ApplicationEventPublisher实例。可以通过如下两种方式为我们的业务对象注入ApplicationEventPublisher的依赖。

- 使用ApplicationEventPublisherAware接口。在ApplicationContext类型的容器启动时，会自动识别该类型的bean定义并将ApplicationContext容器本身作为ApplicationEventPublisher注入当前对象，而ApplicationContext容器本身就是一个ApplicationEventPublisher。
- 使用ApplicationContextAware接口。既然ApplicationContext本身就是一个ApplicationEventPublisher，那么通过ApplicationContextAware几乎达到第一种方式相同的效果。

下面，我们把之前的MethodExecutionEvent相关类改装一下，也好看看改装成使用容器内的事件发布到底是个什么样子。

1. MethodExecutionEvent的改装

因为ApplicationListener只通过void onApplicationEvent(ApplicationEvent event)这一个事件处理方法来处理事件，所以现在要在事件类中尽量保存必要的信息。改装后的MethodExecutionEvent类定义如代码清单5-16所示。

代码清单5-16 改装后的MethodExecutionEvent类定义

```
public class MethodExecutionEvent extends ApplicationEvent {
  private static final long serialVersionUID = -71960369269303337L;
  private String methodName;
  private MethodExecutionStatus methodExecutionStatus;

  public MethodExecutionEvent(Object source) {
    super(source);
  }
  public MethodExecutionEvent(Object source,String methodName, ➥
  MethodExecutionStatus methodExecutionStatus)
  {
    super(source);
    this.methodName = methodName;
    this.methodExecutionStatus = methodExecutionStatus;
  }
  public String getMethodName() {
    return methodName;
  }
  public void setMethodName(String methodName) {
    this.methodName = methodName;
  }
  public MethodExecutionStatus getMethodExecutionStatus() {
    return methodExecutionStatus;
  }
  public void setMethodExecutionStatus(MethodExecutionStatus methodExecutionStatus) {
    this.methodExecutionStatus = methodExecutionStatus;
```

```
    }
}
```

2. **MethodExecutionEventListener**

我们的MethodExecutionEventListener不再是接口，而是具体的ApplicationListener实现类。因为ApplicationListener已经取代了MethodExecutionEventListener原来的角色，所以，改装后的MethodExecutionEventListener定义如下：

```
public class MethodExecutionEventListener implements ApplicationListener {

    public void onApplicationEvent(ApplicationEvent evt) {
        if(evt instanceof MethodExecutionEvent)
        {
            // 执行处理逻辑
        }
    }
}
```

3. **MethodExeuctionEventPublisher改造**

MethodExeuctionEventPublisher改造后如代码清单5-17所示。

代码清单5-17　改造后的MethodExeuctionEventPublisher定义

```
public class MethodExeuctionEventPublisher implements ApplicationEventPublisherAware {
    private ApplicationEventPublisher eventPublisher;

    public void methodToMonitor()
    {
        MethodExecutionEvent beginEvt = new ➥
        MethodExecutionEvent(this,"methodToMonitor",MethodExecutionStatus.BEGIN);
        this.eventPublisher.publishEvent(beginEvt);
        // 执行实际方法逻辑
        // ...
        MethodExecutionEvent endEvt = new ➥
        MethodExecutionEvent(this,"methodToMonitor",MethodExecutionStatus.END);
        this.eventPublisher.publishEvent(endEvt);
    }

    public void setApplicationEventPublisher(ApplicationEventPublisher appCtx) {
        this.eventPublisher = appCtx;
    }
}
```

现在，直接使用注入的eventPublisher来发布事件，而不用自己实现事件发布逻辑了。需要注意的就是，我们实现了ApplicationEventPublisherAware接口（当然，ApplicationContextAware也是可以的）。

4. **注册到ApplicationContext容器**

最后一步工作就是将MethodExeuctionEventPublisher和MethodExecutionEventListener注册到ApplicationContext容器中。当MethodExeuctionEventPublisher的methodToMonitor方法被调用时，事件即被发布。配置如下所示：

```
<bean id="methodExecListener" class="...MethodExecutionEventListener">
</bean>

<bean id="evtPublisher" class="...MethodExeuctionEventPublisher">
</bean>
```

整个改造就此宣告结束！

> **提示**　你可能觉得我们的实例没有任何实际意义。不过，我想提醒的是，如果你尝试在每次发布事件的时候，将当前系统时间或者其他信息也通过MethodExecutionEvent传给具体的ApplicationListener处理，情况是否有所改观呢？你完全可以通过这样的方式来监控系统性能了！要知道，完全可以在这个的基础上往简单的AOP迈进哦！

Spring的容器内事件发布机制初步看来无法脱离容器单独使用。不过，要想做，也不是不可以的，只不过是直接使用ApplicationEventMulticaster接口进行事件发布而已。就提这一句吧，你如果有兴趣可以自己尝试一下。

5.4　多配置模块加载的简化

实际上，这也不算ApplicationContext比较突出的特色功能，只是相对于BeanFactory来说，在这一点上做得更好罢了。

我们知道，在使用Spring的IoC轻量级容器进行实际开发的过程中，为了避免出现整个团队因某个资源独占而无法并行、高效地完成工作等问题，通常会将整个系统的配置信息按照某种关注点进行分割，使得关注点逻辑良好地划分到不同的配置文件中，如按照功能模块或者按照系统划分的层次等。这样，在加载整个系统的bean定义时，就需要让容器同时读入划分到不同配置文件的信息。相对于BeanFactory来说，ApplicationContext大大简化了这种情况下的多配置文件的加载工作。

假设在文件系统中存在多个Spring的配置文件，它们所在路径如下所示：

```
{user.dir}/conf/dao-tier.springxml
{user.dir}/conf/view-tier.springxml
{user.dir}/conf/business-tier.springxml
...
```

通过ApplicationContext，我们只要以String[]形式传入这些配置文件所在的路径，即可构造并启动容器，如代码清单5-18所示。

代码清单5-18　使用ApplicationContext加载多个配置文件

```
String[] locations = new String[]{ "conf/dao-tier.springxml", ➥
"conf/view-tier.springxml", "conf/business-tier.springxml"};
ApplicationContext container = new FileSystemXmlApplicationContext(locations);
// 或者
ApplicationContext container = new ClassPathXmlApplicationContext(locations);
...

甚至于使用通配符
ApplicationContext container = new FileSystemXmlApplicationContext("conf/**/*.springxml");
...
```

而使用BeanFactory来加载这些配置，则需要动用过多的代码，如以下代码所示：

```
BeanFactory parentFactory = new XmlBeanFactory➥
(new FileSystemResource("conf/dao-tier.springxml"));
BeanFactory subFactory = new XmlBeanFactory➥
(new FileSystemResource("conf/view-tier.springxml"),parentFactory);
BeanFactory subsubFactory = new XmlBeanFactory➥
(new FileSystemResource("conf/business-tier.springxml"),subFactory);
...
```

```
BeanFactory container = new XmlBeanFactory(new FileSystemResource("..."),sub...Factory);
...
```

当然，我只是故意给出了一个比较损的对比。实际上，如果通过在某一个主配置文件中使用<import>分别加载其余的配置文件，然后容器就可以通过加载这个主配置文件，来加载其他的配置文件了。但使用<import>的问题，在于需要时刻关注主配置文件与其他配置文件的一致性。

除了可以批量加载配置文件之外，ClassPathXmlApplicationContext还可以通过指定Classpath中的某个类所处位置来加载相应配置文件，配置文件分布结构如下（例子来自Spring参考文档）：

```
com/
  foo/
    services.xml
    daos.xml
    MessengerService.class
```

ClassPathXmlApplicationContext可以通过MessengerService类在Classpath中的位置定位配置文件，而不用指定每个配置文件的完整路径名，如以下代码所示：

```
ApplicationContext ctx = new ClassPathXmlApplicationContext(
  new String[] {"services.xml", "daos.xml"}, MessengerService.class);
```

读者可以参考各ApplicationContext实现类的Javadoc取得更多相关信息。

5.5　小结

ApplicationContext是Spring在BeanFactory基础容器之上，提供的另一个IoC容器实现。它拥有许多BeanFactory所没有的特性，包括统一的资源加载策略、国际化信息支持、容器内事件发布以及简化的多配置文件加载功能。本章对ApplicationContext的这些新增特性进行了详尽的阐述。希望读者在学习完本章内容之后，对每一种特性的来龙去脉都能了如指掌。

第6章

Spring IoC容器之扩展篇

6

本章内容

- Spring 2.5的基于注解的依赖注入
- Spring 3.0展望

6.1 Spring 2.5 的基于注解的依赖注入

Spring 2.5提供的基于注解的依赖注入功能延续了Spring框架内在IoC容器设计与实现上的一致性。除了依赖关系的"表达"方式上的不同，底层的实现机制基本上保持一致。如果我们已经从Spring的IoC容器的XML之旅中成功走过来，那么在体验基于注解的依赖注入的过程中，一定会发现许多似曾相识的身影。你瞧，基于XML配置方式的自动绑定功能，就是我们再次邂逅的第一位老朋友……

6.1.1 注解版的自动绑定（@Autowired）

1. 从自动绑定（autowire）到@Autowired

在使用依赖注入绑定FXNews相关实现类时，为了减少配置量，我们可以采用Spring的IoC容器提供的自动绑定功能，如下所示：

```
<beans default-autowire="byType">
   <bean id="newsProvider"    class="..FXNewsProvider" autowire="byType"/>
   <bean id="djNewsListener"  class="..DowJonesNewsListener"/>
   <bean id="djNewsPersister" class="..DowJonesNewsPersister"/>
</beans>
```

可以通过default-autowire来指定默认的自动绑定方式，也可以通过每个bean定义上的autowire来指定每个bean定义各自的自动绑定方式，它们都是触发容器对相应对象给予依赖注入的标志。而将自动绑定的标志用注解来表示时，也就得到了基于注解的依赖注入，或者更确切地称为基于注解的自动绑定。

@Autowired是基于注解的依赖注入的核心注解，它的存在可以让容器知道需要为当前类注入哪些依赖。比如可以使用@Autowired对FXNewsProvider类进行标注，以表明要为FXNewsProvider注入的依赖。代码清单6-1给出了标注后的情况。

代码清单6-1　使用@Autowired标注后的FXNewsProvider

```
public class FXNewsProvider
{
   private IFXNewsListener  newsListener;
   private IFXNewsPersister newPersistener;
```

```
    @Autowired
    public FXNewsProvider(IFXNewsListener newsListner,IFXNewsPersister newsPersister)
    {
      this.newsListener   = newsListner;
      this.newPersistener = newsPersister;
    }
    ...
}
```

与原有的byType类型的自动绑定方式类似，@Autowired也是按照类型匹配进行依赖注入的，只不过，它要比byType更加灵活，也更加强大。@Autowired可以标注于类定义的多个位置，包括如下几个。

域（Filed）或者说属性（Property）。不管它们声明的访问限制符是private、protected还是public，只要标注了@Autowired，它们所需要的依赖注入需求就都能够被满足，如下所示：

```
public class FXNewsProvider
{
    @Autowired
    private IFXNewsListener  newsListener;
    @Autowired
    private IFXNewsPersister newPersistener;
    ...
}
```

构造方法定义（Constructor）。标注于类的构造方法之上的@Autowired，相当于抢夺了原有自动绑定功能中“constructor”方式的权利，它将根据构造方法参数类型，来决定将什么样的依赖对象注入给当前对象。从最初的代码示例中，我们可以看到标注于构造方法之上的@Autowired的用法。

方法定义（Method）。@Autowired不仅可以标注于传统的setter方法之上，而且还可以标注于任意名称的方法定义之上，只要该方法定义了需要被注入的参数。代码清单6-2给出了一个标注于这种任意名称方法之上的@Autowired使用示例代码。

代码清单6-2　标注于方法之上的@Autowired代码示例

```
public class FXNewsProvider
{
    private IFXNewsListener  newsListener;
    private IFXNewsPersister newPersistener;

    @Autowired
    public void setUp(IFXNewsListener  newsListener,IFXNewsPersister newPersistener)
    {
      this.newsListener   = newsListener;
      this.newPersistener = newPersistener;
    }
    ...
}
```

现在，虽然可以随意地在类定义的各种合适的地方标注@Autowired，希望这些被@Autowired标注的依赖能够被注入，但是，仅将@Autowired标注于类定义中并不能让Spring的IoC容器聪明到自己去查看这些注解，然后注入符合条件的依赖对象。容器需要某种方式来了解，哪些对象标注了@Autowired，哪些对象可以作为可供选择的依赖对象来注入给需要的对象。在考虑使用什么方式实现这一功能之前，我们先比较一下原有的自动绑定功能与使用@Autowired之后产生了哪些差别。

使用自动绑定的时候，我们将所有对象相关的bean定义追加到了容器的配置文件中，然后使用default-autowire或者autowire告知容器，依照这两种属性指定的绑定方式，将容器中各个对象绑

定到一起。在使用@Autowired之后，default-autowire或者autowire的职责就转给了@Autowired，所以，现在，容器的配置文件中就只剩下了一个个孤伶伶的bean定义，如下所示：

```
<beans>
  <bean id="newsProvider"    class="..FXNewsProvider"/>
  <bean id="djNewsListener"  class="..DowJonesNewsListener"/>
  <bean id="djNewsPersister" class="..DowJonesNewsPersister"/>
</beans>
```

为了给容器中定义的每个bean定义对应的实例注入依赖，可以遍历它们，然后通过反射，检查每个bean定义对应的类上各种可能位置上的@Autowired。如果存在的话，就可以从当前容器管理的对象中获取符合条件的对象，设置给@Autowired所标注的属性域、构造方法或者方法定义。整个逻辑如代码清单6-3中的原型代码所示。

代码清单6-3 容器遍历@Autowired并进行依赖注入的原型代码示例

```
Object[] beans = ...;
for(Object bean:beans)
{
  if(autowiredExistsOnField(bean))
  {
    Field f = getQulifiedField(bean));
    setAccessiableIfNecessary(f);
    f.set(getBeanByTypeFromContainer());
  }
  if(autowiredExistsOnConstructor(bean))
  {
    ...
  }
  if(autowiredExistsOnMethod(bean))
  {
    ...
  }
}
```

看到以上的原型代码所要完成的功能以及我们的设想，你一定想到了，我们可以提供一个Spring的IoC容器使用的BeanPostProcessor自定义实现，让这个BeanPostProcessor在实例化bean定义的过程中，来检查当前对象是否有@Autowired标注的依赖需要注入。org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor就是Spring提供的用于这一目的的BeanPostProcessor实现。所以，很幸运，我们不用自己去实现它了。

将FXNews相关类定义使用@Autowired标注之后，只要在IoC容器的配置文件中追加AutowiredAnnotationBeanPostProcessor就可以让整个应用开始运作了，如下所示：

```
<beans>
  <bean class="org.springframework.beans.factory.annotation. ➥
  AutowiredAnnotationBeanPostProcessor"/>

  <bean id="newsProvider"    class="..FXNewsProvider"/>
  <bean id="djNewsListener"  class="..DowJonesNewsListener"/>
  <bean id="djNewsPersister" class="..DowJonesNewsPersister"/>
</beans>
```

当然，这需要我们使用ApplicationContext类型的容器，否则还得做点儿多余的准备工作。

> **注意**　看着依赖注入相关的信息，一半分散在Java源代码中（@Autowired标注的信息），一半依然留在XML配置文件里，你心里一定觉得很不爽。实际上，我也是，这不是折腾人吗？不过，别急，让我们先解决眼前的另一个问题，稍后再回过头来看看怎么进一步统一这两片国土。

2. @Qualifier的陪伴

@Autowired是按照类型进行匹配，如果当前@Autowired标注的依赖在容器中只能找到一个实例与之对应的话，那还好。可是，要是能够同时找到两个或者多个同一类型的对象实例，又该怎么办呢？我们自己当然知道应该把具体哪个实例注入给当前对象，可是，IoC容器并不知道，所以，得通过某种方式告诉它。这时，就可以使用@Qualifier对依赖注入的条件做进一步限定，使得容器不再迷茫。

@Qualifier实际上是byName自动绑定的注解版，既然IoC容器无法自己从多个同一类型的实例中选取我们真正想要的那个，那么我们不妨就使用@Qualifier直接点名要哪个好了。假设FXNews-Provider使用的IFXNewsListener有两个实现，一个是DowJonesNewsListener，一个是ReutersNe-wsListener，二者相关配置如下：

```
<beans>
   <bean class="org.springframework.beans.factory.annotation. ➥
   AutowiredAnnotationBeanPostProcessor"/>

   <bean id="newsProvider"     class="..FXNewsProvider"/>
   <bean id="djNewsListener"   class="..DowJonesNewsListener"/>
   <bean id="reutersNewsListner" class="..ReutersNewsListener"/>
   <bean id="djNewsPersister" class="..DowJonesNewsPersister"/>
</beans>
```

如果我们想让FXNewsProvider使用ReutersNewsListener，那么就可以在FXNewsProvider的类定义中使用@Qualifier指定这一选择结果，如下：

```
public class FXNewsProvider
{
   @Autowired
   @Qualifier("reutersNewsListner")
   private IFXNewsListener  newsListener;
   @Autowired
   private IFXNewsPersister newPersistener;
   ...
}
```

以上我们使用的是标注于属性域的@Autowired进行依赖注入。如果使用@Autowired来标注构造方法或者方法定义的话，同样可以使用@Qualifier标注方法参数来达到限定注入实例的目的。代码清单6-4给出的正是标注于方法参数之上的@Qualifier的使用示例。

代码清单6-4　标注于方法参数之上的@Qualifier

```
public class FXNewsProvider
{
   ...

   @Autowired
   public void setUp(@Qualifier("reutersNewsListner") IFXNewsListener ➥
   newsListener,IFXNewsPersister newPersistener)
   {
     this.newsListener    = newsListener;
     this.newPersistener = newPersistener;
```

```
    }
    ...
}
```

除此之外，@Qualifier还可以用于标注注解类型，这主要用于自定义@Qualifier的场合。有关自定义@Qualifier的内容我们这里就不做赘述了，Spring 2.5的参考文档中有详细的介绍。

6.1.2 @Autowired之外的选择——使用JSR250标注依赖注入关系

Spring 2.5提供的基于注解的依赖注入，除了可以使用Spring提供的@Autowired和@Qualifier来标注相应类定义之外，还可以使用JSR250的@Resource和@PostConstruct以及@PreDestroy对相应类进行标注，这同样可以达到依赖注入的目的。

@Resource与@Autowired不同，它遵循的是byName自动绑定形式的行为准则，也就是说，IoC容器将根据@Resource所指定的名称，到容器中查找beanName与之对应的实例，然后将查找到的对象实例注入给@Resource所标注的对象。同样的FXNewsProvider，如若使用@Resource进行标注以获取依赖注入的话，类似如下的样子：

```
public class FXNewsProvider
{
    @Resource(name="djNewsListener")
    private IFXNewsListener  newsListener;
    @Resource(name="djNewsPersister")
    private IFXNewsPersister newPersistener;
    ...
}
```

JSR250规定，如果@Resource标注于属性域或者方法之上的话，相应的容器将负责把指定的资源注入给当前对象，所以，除了像我们这样直接在属性域上标注@Resource，还可以在构造方法或者普通方法定义上标注@Resource，这与@Autowired能够存在的地方大致相同。

确切地说，@PostConstruct和@PreDestroy不是服务于依赖注入的，它们主要用于标注对象生命周期管理相关方法，这与Spring的InitializingBean和DisposableBean接口，以及配置项中的init-method和destroy-method起到类似的作用。代码清单6-5给出了可能使用这两个注解的示例代码。

代码清单6-5 使用@PostConstruct和@PreDestroy标注对象的生命周期管理方法

```
public class LifecycleEnabledClass
{
    @PostConstruct
    public void setUp()
    {
        ...
    }
    @PreDestroy
    public void destroy()
    {
        ...
    }
}
```

如果想某个方法在对象实例化之后被调用，以做某些准备工作，或者想在对象销毁之前调用某个方法清理某些资源，那么就可以像我们这样，使用@PostConstruct和@PreDestroy来标注这些方法。当然，是使用@PostConstruct和@PreDestroy，还是使用Spring的InitializingBean和Disposable-

Bean接口，或者init-method和destroy-method配置项，可以根据个人的喜好自己决定。

天上永远不会掉馅饼，我们只是使用@Resource或者@PostConstruct和@PreDestroy标注了相应对象，并不能给该对象带来想要的东西。所以，就像@Autowired需要AutowiredAnnotationBean-PostProcessor为它与IoC容器牵线搭桥一样，JSR250的这些注解也同样需要一个BeanPost-Processor帮助它们实现自身的价值。这个BeanPostProcessor就是org.springframework.context.annotation.CommonAnnotationBeanPostProcessor，只有将CommonAnnotationBeanPostProcessor添加到容器，JSR250的相关注解才能发挥作用，通常如下添加相关配置即可：

```
<beans>
<bean class="org.springframework.context.annotation.CommonAnnotationBeanPostProcessor"/>

<bean id="newsProvider"    class="..FXNewsProvider"/>
<bean id="djNewsListener"  class="..DowJonesNewsListener"/>
<bean id="djNewsPersister" class="..DowJonesNewsPersister"/>
</beans>
```

既然不管是@Autowired还是@Resource都需要添加相应的BeanPostProcessor到容器，那么我们就可以在基于XSD的配置文件中使用一个<context:annotation-config>配置搞定以上所有的BeanPostProcessor配置，如代码清单6-6所示。

代码清单6-6　使用<context:annotation-config/>激活注解的相关功能

```
<beans xmlns="http://www.springframework.org/schema/beans" ➥
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ➥
xmlns:context="http://www.springframework.org/schema/context" ➥
xmlns:p="http://www.springframework.org/schema/p" ➥
xsi:schemaLocation=" ➥
http://www.springframework.org/schema/beans ➥
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd ➥
http://www.springframework.org/schema/context ➥
http://www.springframework.org/schema/context/spring-context-2.5.xsd"> ➥

   <context:annotation-config/>

   <bean id="newsProvider"    class="..FXNewsProvider"/>
   <!--其他bean定义-->
   ...
</beans>
```

<context:annotation-config>不但帮我们把AutowiredAnnotationBeanPostProcessor和CommonAnnotationBeanPostProcessor注册到容器，同时还会把PersistenceAnnotationBeanPost-Processor和RequiredAnnotationBeanPostProcessor一并进行注册，可谓一举四得啊！

> **注意**　Spring提供的@Autowired加上@Qualifier和JSR250提供的@Resource等注解属于两个派系。如果要实现依赖注入的话，使用一个派别的注解就可以了。当然，既然<context:anno-tation-config>对两个派系都提供了BeanPostProcessor的支持，混合使用也是没有问题的，只要别造成使用上的混乱就行。

6.1.3　将革命进行得更彻底一些（classpath-scanning 功能介绍）

好了，该来解决让我们不爽的那个问题了。到目前为止，我们还是需要将相应对象的bean定义，

一个个地添加到IoC容器的配置文件中。与之前唯一的区别就是，不用在配置文件中明确指定依赖关系了（改用注解来表达了嘛）。既然使用注解来表达对象之间的依赖注入关系，那为什么不搞的彻底一点儿，将那些几乎“光秃秃”的bean定义从配置文件中彻底消灭呢？OK，我们想到了，Spring开发团队也想到了，classpath-scanning的功能正是因此而诞生的！

使用相应的注解对组成应用程序的相关类进行标注之后，classpath-scanning功能可以从某一顶层包（base package）开始扫描。当扫描到某个类标注了相应的注解之后，就会提取该类的相关信息，构建对应的BeanDefinition，然后把构建完的BeanDefinition注册到容器。这之后所发生的事情就不用我说了，既然相关的类已经添加到了容器，那么后面BeanPostProcessor为@Autowired或者@Resource所提供的注入肯定是有东西拿咯！

classpath-scanning功能的触发是由<context:component-scan>决定的。按照如下代码，在XSD形式（也只能是XSD形式）的配置文件中添加该项配置之后，classpath-scanning功能立即开启：

```
<beans xmlns="http://www.springframework.org/schema/beans" ➥
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ➥
xmlns:context="http://www.springframework.org/schema/context" ➥
xsi:schemaLocation="http://www.springframework.org/schema/beans ➥
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd ➥
http://www.springframework.org/schema/context ➥
http://www.springframework.org/schema/context/spring-context-2.5.xsd">
<context:component-scan base-package="org.spring21"/>
</beans>
```

现在<context:component-scan>将遍历扫描org.spring21路径下的所有类型定义，寻找标注了相应注解的类，并添加到IoC容器。

> **提示** 如果要扫描的类定义存在于不同的源码包下面，也可以为base-package指定多个以逗号分隔的扫描路径。需要的话，不要犹豫！

<context:component-scan>默认扫描的注解类型是@Component。不过，在@Component语义基础上细化后的@Repository、@Service和@Controller也同样可以获得<context:component-scan>的青睐。@Component的语义更广、更宽泛，而@Repository、@Service和@Controller的语义则更具体。所以，同样对于服务层的类定义来说，使用@Service标注它，要比使用@Component更为确切。对于其他两种注解也是同样道理，我们暂且使用语义更广的@Component来标注FXNews相关类，以便摆脱每次都要向IoC容器配置添加bean定义的苦恼。使用@Component标注后的FXNews相关类见代码清单6-7。

代码清单6-7 使用@Component标注后的FXNews相关类定义

```
@Component
public class FXNewsProvider
{
  @Autowired
  private IFXNewsListener  newsListener;
  @Autowired
  private IFXNewsPersister newPersistener;
  ...
}

@Component("djNewsListener")
public class DowJonesNewsListener implements IFXNewsListener
```

```
{
  ...
}

@Component
public class DowJonesNewsPersister implements IFXNewsPersister
{
  ...
}
```

<context:component-scan>在扫描相关类定义并将它们添加到容器的时候，会使用一种默认的命名规则，来生成那些添加到容器的bean定义的名称（beanName）。比如DowJonesNewsPersister通过默认命名规则将获得dowJonesNewsPersister作为bean定义名称。如果想改变这一默认行为，就可以像以上DowJonesNewsListener所对应的@Component那样，指定一个自定义的名称[①]。

现在，除了<context:component-scan>是唯一需要添加到IoC容器的配置内容，所有的工作都可以围绕着使用注解的Java源代码来完成了。如果现在加载配置文件，启动FXNewProvider来处理外汇新闻的话，我们可以得到预期的运行效果[②]，运行的代码如下所示：

```
ApplicationContext ctx = new ClassPathXmlApplicationContext("../conf.xml");
FXNewProvider provider = (FXNewProvider)ctx.getBean("FXNewsProvider");
provider.getAndPersistNews();
```

你或许会觉得有些诧异，因为我们并没有使用<context:annotation-config>甚至直接将相应的BeanPostProcessor添加到容器中，而FXNewsProvider怎么会获得相应的依赖注入呢？这个得怪<context:component-scan>“多管闲事”，它同时将AutowiredAnnotationBeanPostProcessor和CommonAnnotationBeanPostProcessor一并注册到了容器中，所以，依赖注入的需求得以满足。如果你不喜欢，非要自己通过<context:annotation-config>或者直接添加相关BeanPost-Processor的方式来满足@Autowired或者@Resource的需求，可以将<context:component-scan>的annotation-config属性值从默认的true改为false。不过，我想没有太好的理由非要这么做吧？

<context:component-scan>的扫描行为可以进一步定制，默认情况下它只关心@Component、@Repository、@Service和@Controller四位大员，但我们可以丰富这一范围，或者对默认的扫描结果进行过滤以排除某些类，<context:component-scan>的嵌套配置项可以帮我们达到这一目的。代码清单6-8演示了<context:component-scan>部分嵌套配置项的使用。

代码清单6-8　<context:component-scan>部分嵌套配置项的使用示例

```
<beans xmlns="http://www.springframework.org/schema/beans" ➥
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ➥
xmlns:context="http://www.springframework.org/schema/context" ➥
xsi:schemaLocation="http://www.springframework.org/schema/beans ➥
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd ➥
http://www.springframework.org/schema/context ➥
http://www.springframework.org/ schema/context/spring-context-2.5.xsd">
<context:component-scan base-package="org.spring21">
```

① 另一种方法就是使用自定义实现的BeanNameGenerator，通过<context:component-scan>的name-generator属性指定我们自己的BeanNameGenerator实现类来替换掉默认的BeanNameGenerator，也可以改变默认的bean定义名称生成规则。

② 注意，我们通过FXNewsProvider作为bean定义名称来获取FXNewsProvider的实例，对于开头都是大写的类名来说，bean定义名称实际上就相当于类名，这与默认的命名生成规则有些小小的差异。

```
        <context:include-filter type="annotation" ➥
        expression="cn.spring21.annotation.FXService"/>
        <context:exclude-filter type="aspectj" expression=".."/>
    </context:component-scan>
</beans>
```

include-filter和exclude-filter可以使用的type类型有annotation、assignable、regex和aspectj四种。它们的更多信息可以参考最新的Spring 2.5参考文档。上例中，我们增加了@FXService作为新的被扫描注解对象，并使用aspectj表达式排除某些扫描结果。

注意 有关基于注解的依赖注入和classpath-scanning功能的更多细节不能尽述，可以对照Spring 2.5的参考文档做进一步的认识。

6.2 Spring 3.0 展望

Java 5是Java平台发展史上的一个里程碑，它为Java平台带来了诸如泛型（Generics）[1]、注解等新的特性。随着时间推移，Java 5将逐渐流行并成为各种Java应用所需要的“最低”配置。Spring 3.0的发布将顺应这种形势，对现有的API进行全线升级。无疑，现有代码库（codebase）将接受一个大手术。届时，Spring框架将会“脱筋换骨”，以全新的面貌展现在我们的面前。

从发布的Spring 3.0 M2版本中，我们可以发现许多类都已经泛型化，像FactoryBean、ActionListener等类。

```
public interface FactoryBean<T> {
    T getObject() throws Exception;
    Class<? extends T> getObjectType();
    boolean isSingleton();
}

public interface ApplicationListener<E extends ApplicationEvent> extends EventListener {
    void onApplicationEvent(E event);

}
```

当然，Java 5风格的API升级并不限于泛型这一点，根据Spring团队博客上的介绍，他们还有可能在Spring 3.0中追加“基于注解的工厂方法（Annotation-based Factory method）”。虽然“基于注解的工厂方法”这一功能还未发布，不过我们可以先设想一下，如果要我们来实现，应该如何做呢？

首先，Spring 3.0将追加一个注解定义，用来标注相应的工厂方法。我们暂且假设该注解定义如下：

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface FactoryMethod {

}
```

@FactoryMethod看起来有些“单薄”。不过没关系，它只是为了帮助我们构建一个代码原型，以便说明问题。在有了这一注解之后，我们就可以标注相应的工厂方法，比如：

```
public class AnyFactory {
```

① generics有的翻译成“模板”，有的翻译成“泛型”。如果硬要翻译的话，我倾向于“泛型”，因为“模板”已经有了其他的含义，很容易引起混淆以及交流障碍。

```
    @FactoryMethod
    public <T> T create(Class<T> clazz)
    {
      // 创建并返回相应的实例
      try {
        return clazz.newInstance();
      } catch (InstantiationException e) {
        //...
        return null;
      } catch (IllegalAccessException e) {
        //...
        return null;
      }
    }
}
```

但是，只使用@FactoryMethod标注相应方法，Spring并不会知道这个被标注的方法就是一个工厂方法。我们还需要通过某种方式通知Spring说，“标注了@FactoryMethod的方法定义都是需要特殊对待的工厂方法”，或者“如果哪个类中有方法定义被标注了@FactoryMethod，那么应该对这个类做一些特殊处理”。在有了classpath-scanning这一“基础设施”之后，要实现这样的需求只需要在classpath-scanning期间添加相应的处理逻辑即可。要完成这一工作可能有如下几种选择。

- classpath-scanning的时候，在对标注了@Component或者其他注解的类进行处理的同时，也稍带检查一下当前类是否也标注了@FactoryMethod。如果是，则进一步将工厂方法相关信息添加到相应的BeanDefinition。
- 向容器中新添加一个BeanFactoryPostProcessor实现类，由这个BeanFactoryPostProcessor来统一做@FactoryMethod的相关处理。不过，这需要org.springframework.beans.factory.config.BeanFactoryPostProcessor接口开放更多一些的接口。在Spring 3.0 M2的Javadoc中可以发现这样的迹象，所以这条路应该也可以走通。

还可以尝试其他思路。或者等Spring 3.0发布之后自己验证一下是不是这么回事……

6.3　小结

Spring最初并不支持基于注解的依赖注入方式。所以，在Spring 2.5中引入这一依赖注入方式的时候，肯定要在维护整个框架设计与实现的一致性和引入这种依赖注入方式对整个框架的冲击之间做出权衡。最终的结果我们已经看到了，Spring 2.5中引入的基于注解的依赖注入从整体上保持了框架内的一致性，同时又提供了足够的基于注解的依赖注入表达能力。我想，最初的决定和最终的效果都是令人满意的。虽然我们还会部分地依赖于容器的配置文件，但通过20%的工作却可以带来80%的效果，这本身已经是最好的结果了。

不过，从实际开发角度看，如果非要使用完全基于注解的依赖注入的话，或许会遇到一些过不去的坎儿。比如，对于第三方提供的类库，肯定没法给其中的相关类标注@Component之类的注解。这时，我们可以结合使用基于配置文件的依赖注入方式。毕竟，基于XML的依赖注入方式是Spring提供的最基本、也最为强大的表达方式了！

到目前为止，我们已经结束了Spring IoC容器的旅程。接下来的第三部分将带领读者探索Spring AOP框架的精彩世界。

Part 3 第三部分

Spring AOP 框架

在真正进入Spring的AOP框架的“内心世界”之前，有必要先讲述明白整个AOP的概念以及它的来龙去脉。毕竟，如果对一般情况都能了如指掌，那么理解特殊的AOP实现产品就不会太困难了。

本部分内容

- 第 7 章　一起来看 AOP
- 第 8 章　Spring AOP 概述及其实现机制
- 第 9 章　Spring AOP 一世
- 第 10 章　Spring AOP 二世
- 第 11 章　AOP 应用案例
- 第 12 章　Spring AOP 之扩展篇

第7章

一起来看AOP

本章内容

- AOP的尴尬
- AOP走向现实
- Java平台上的AOP实现机制
- AOP国家的公民

软件开发一直在寻求更加高效、更易维护甚至更易扩展的方式。为了提高开发效率，我们对开发使用的语言进行抽象，走过了从汇编时代到现在各种高级语言繁盛之时期；为了便于维护和扩展，我们就对某些相同的功能进行归类并使之模块化，冲出了最初的“原始部落”，走过了从过程化编程（Procedural Programming）到面向对象编程（OOP，Object-Oriented Programming）的“短暂而漫长”的历程。但不管走过的路有多长，多么坎坷，我们一直没有放弃寻找更加完美、更加高效的软件开发方法。过去如此，现在亦然。

当OOP/OOSD（Object-Oriented Software Development）被提出来，以取代①过去的基于过程化编程的开发方法的时候，或许那个时代的人都会以为，面向对象编程和面向对象的软件开发（OOP/OOSD）就是我们一直所追求的那颗能够搞定一切的“银弹”。但不得不承认的是，即使面向对象的软件开发模式，依然不能很好地解决软件开发中的所有问题。

软件开发的目的，最终是为了解决各种需求，包括业务需求和系统需求。使用面向对象方法，我们可以对业务需求等普通关注点进行很好的抽象和封装，并且使之模块化。但对于系统需求一类的关注点来说，情况却有所不同。

以我曾经参与开发的CREDIT项目为例，它是一个有关贷款业务的管理系统。从业务角度说，该系统提供了顾客贷款申请、顾客信息管理、贷款信息管理、贷款发放回收等功能；这些都属于普通的业务需求。通过面向对象方法，可以很容易地按照功能划分模块并完成开发。图7-1给出了这些功能模块之间清晰的关系。

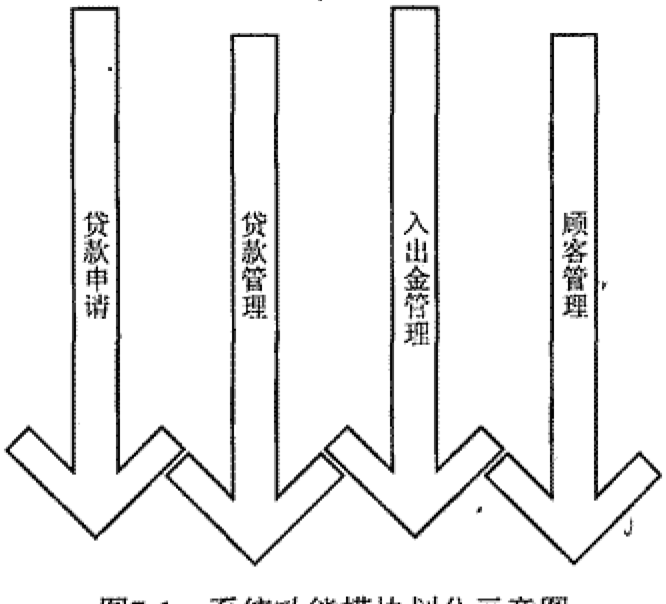

图7-1　系统功能模块划分示意图

对于业务需求而言，需求与其具体实现之间的关系基本上是一对一的。我们可以在系统中某一个确定的点找到针对这种需求的实现，无论从开发还是维护的角度，都比较方便。

---

① 这里的“取代”实际上并非就完全否定过程化编程方法的贡献和一些理念，只是在其基础上更上一层楼。

不过，事情还没完呢！开发中为了调试，或在进入生产环境后为了对系统进行监控，我们需要为这些业务需求的实现对象添加日志记录功能；或者，业务方法的执行需要一定的权限限制。那么方法执行前肯定需要有相应的安全检查功能。而这些则属于系统需求的范畴。虽然需求都很明确（加入日志记录、加入安全检查），但是要将这些需求以面向对象的方式实现并集成到整个的系统中去，可就不是一个需求对应一个实现那么简单了。系统中的每个业务对象都需要加入日志记录，加入相应的安全检查，那么，这些需求的实现代码就会遍及所有业务对象。整个场景如图7-2所示。

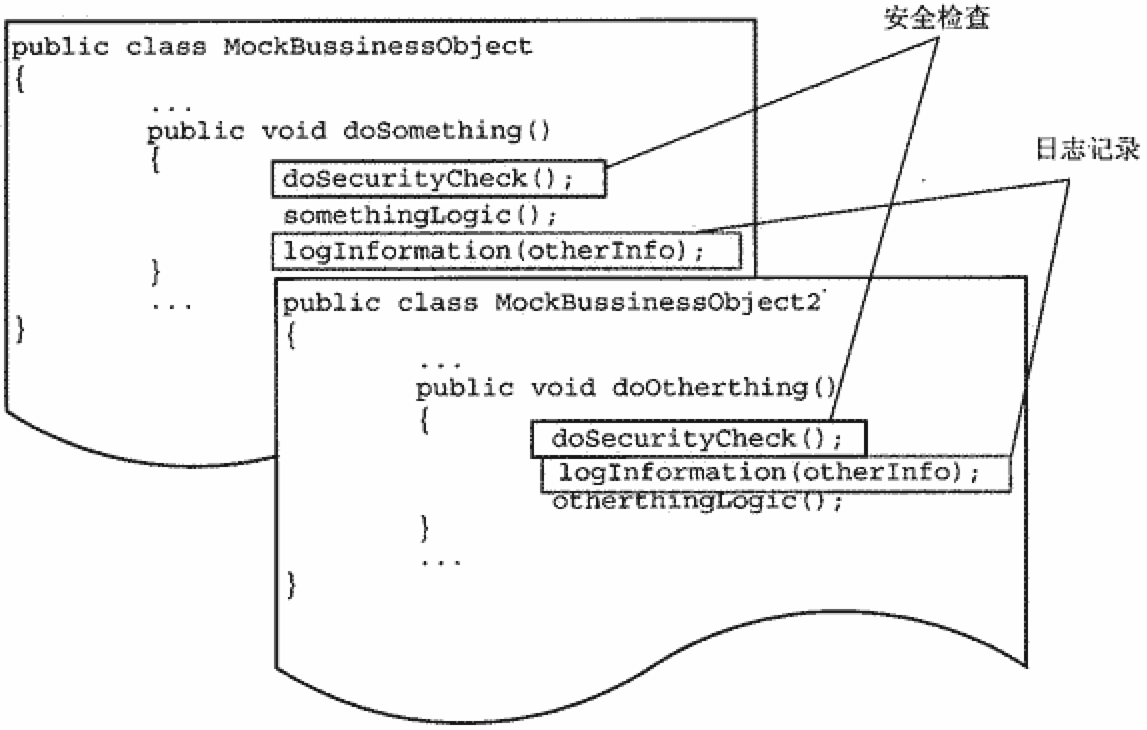

图7-2　日志记录与安全检查等系统需求的生存现状

图7-2中日志记录和安全检查的需求和实现的对应关系还仅仅是1∶2。但随着系统中业务对象的增加，这个对应关系就会变成1∶3、1∶4……1∶100甚至更多。你可以想象一下，随着这个数目的增多，你的系统开发和维护的难度会向一个什么方向发展。

对于系统中普通的业务关注点，OOP可以很好地对其进行分解并使之模块化，但却无法更好地避免类似于系统需求的实现在系统中各处散落这样的问题。所以，我们要寻求一种更好的方法，它可以在OOP的基础上更上一层楼，提出一套全新的方法论来避免以上问题，也可以提供某种方法对基于OOP的开发模式做一个补足，帮助OOP以更好的方式解决以上问题。但迄今为止，我们还找不到比OOP更加有效的软件开发模式。不过，好消息是，我们找到了后者，那就是AOP。

AOP全称为Aspect-Oriented Programming，中文通常翻译为面向方面编程。使用AOP，我们可以对类似于Logging和Security等系统需求进行模块化的组织，简化系统需求与实现之间的对比关系，进而使得整个系统的实现更具模块化。

任何一个软件系统就跟CREDIT系统一样，日志记录、安全检查、事务管理等系统需求就像一把把刀“恶狠狠”地横切到我们组织良好的各个业务功能模块之上（见图7-3）。以AOP的行话来说，这些系统需求是系统中的横切关注点（cross-cutting concern）。使用传统方法，我们无法更好地以模块化的方式，对这些横切关注点进行组织和实现。所以AOP引入了Aspect的概念，用来以模块化的形式对系统中的横切关注点进行封装。Aspect之对于AOP，就相当于Class之对于OOP。我们说过AOP仅是对OOP方法的一种补足，当我们把以Class形式模块化的业务需求和以Aspect形式模块化的系统需求拼

装到一起的时候，整个系统就算完成了。

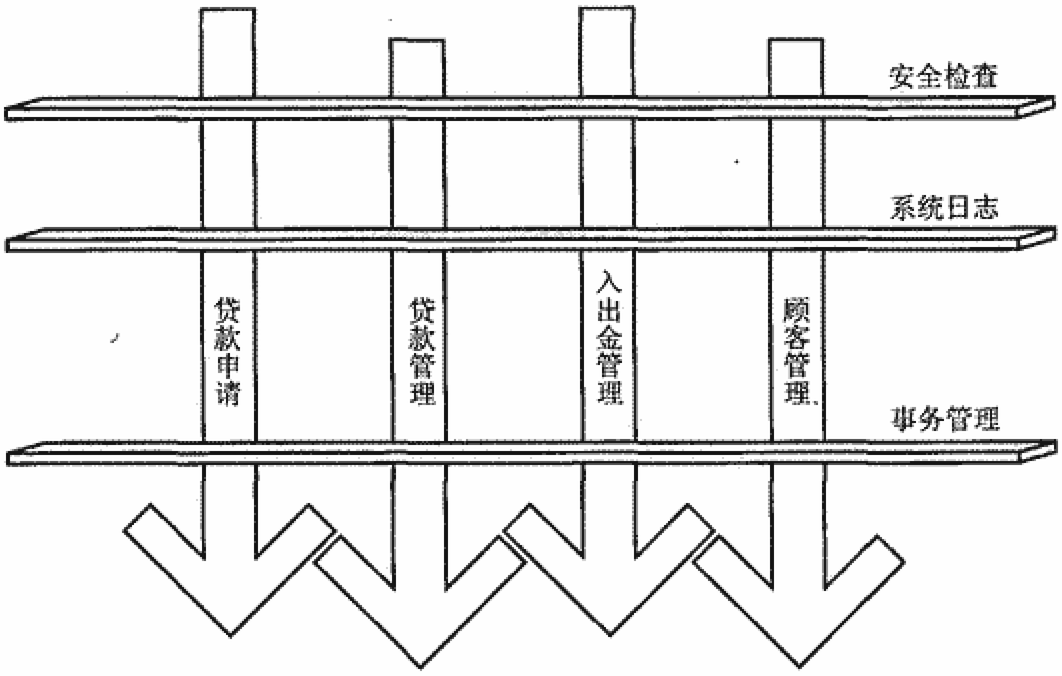

图7-3　加入各种系统需求后的系统模块关系示意图

7.1　AOP 的尴尬

如果把我们的软件系统看作是可以划分为不同形状的积木，对于业务需求类型的积木块和系统需求类型的积木块来说，它们的形状和材质可以是相近甚至是相同的，但摆放的空间位置却完全处于不同的维度。

当对整个系统进行分析之后，我们可以将不同的需求实现为Aspect类型的积木块或者Class类型的积木块。这样，我们就有了如图7-4所示的一盒积木。

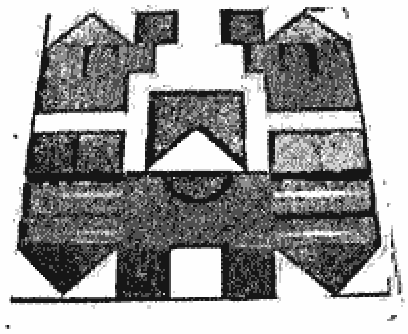

图7-4　未开始搭建的积木

不过，积木块永远是积木块，需要我们动手搭建才能构建出美丽的模型。当我们把Class类型的积木块在一个空间面上摆放，而将Aspect类型的积木块在另一个空间面上摆放的时候，我们就有了一座美丽的城堡（见图7-5），而且最主要的，它是立体的哦！

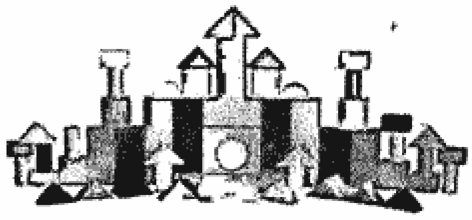

图7-5　搭建完成的积木

也就是说，OOP的空间结合AOP的空间就可以构建一个完美的系统。不过，由于当前技术所限，虽然我们可以构造出AOP使用的各个积木块，但却无法构建属于AOP的独有空间，这就像俄罗斯方块一样。现在我们使用OOP进行软件开发，就好像在玩俄罗斯方块。当各个系统业务模块划分完成之后（俄罗斯方块就那几个特定的砖头形状），剩下的工作，就是在游戏规定的空间维度里，想方设法地把每一层都填满，否则，你就得等着你的系统崩溃，彻底地Game Over。还好，我们已经精熟于那几个砖头的方向调整和恰当的放置位置，就好像我们精熟于面向对象编程的方方面面或者设计模式（Design Pattern）一样。但是，即使如此，也并没有解决根本的问题，我们还是时不时地遗漏某个位置的空间。而实际上，如果允许，最简单的方法就是从我们所处的维度，直接使用合适的砖头，把那

些遗漏的空间填补上就算大功告成。在那个有限的游戏空间内，我们也不用太过于费尽脑汁地考虑如何使用原来的规则消去遗漏的空间块。

俄罗斯方块那个游戏空间就是OOP现在的空间，而AOP应该在另一个空间内，才可以提供最大的便利。但是，正如我们所见，AOP现在没有主权，所以，现时代的AOP实现都需要“寄生”于OOP的主权领土中，系统的维度也依然保持曾经OOP持有的“维度世界纪录”。

7.2 AOP走向现实

AOP是一种理念，要实现这种理念通常需要一种现实的方式。与OOP需要相应的语言支持一样，AOP也需要某种语言以帮助实现相应的概念实体，我们统称这些实现AOP的语言为AOL，即Aspect-Oriented Language，不要跟American On Line混淆哦。

AOL可以与系统实现语言相同，比如，如果系统实现语言为Java，那么，相应的AOL也可以为Java。但AOL并非一定要与系统实现语言相同，它也可以是其他语言，比如AspectJ是扩展自Java的一种AOL，显然与系统实现语言属于不同的两种语言。

除了主流的OOP语言，软件开发界现在已经有一些专门针对AOP的语言扩展。除了上面提到的对Java语言扩展后产生的AspectJ，还有：

- AspectC；
- AspectC++；
- Aspect.Net；
- AspectL(Lisp)；
- AspectPHP。
- ……

囿于现实中AOP技术实现上的尴尬，AOL实现的AOP各个概念实体，最终都需要某种方式集成到系统实现语言所实现的OOP实体组件中。所以，系统实现语言通常称为系统中使用的AOL的“寄生语言”，而将AO组件集成到OOP组件的过程，在AOP中称之为织入（Weave）过程。

将AOP的Aspect织入到OOP系统的实现方式可谓千差万别。但不管如何实现，织入过程是处于AOP和OOP的开发过程之外的，而且对于整个系统的实现是透明的，开发者只需要关注相应的业务需求实现，或者系统需求的实现即可。当所有业务需求和系统需求以模块化的形式开发完成之后，通过织入过程就可以将整个的软件系统集成并付诸使用。

Java界的AOP框架或者说产品，可谓AOP土地上的一朵奇葩，在Xerox公司的PARC（Paro Alto Research Center）提出AOP的一套理论之后，Java业界各种AOP框架就如雨后春笋般涌现，其走过的路亦不可谓不精彩，所以，让我们来回顾一下这段精彩历史何如？

7.2.1 静态AOP时代

静态AOP，即第一代AOP，以最初的AspectJ为杰出代表，其特点是，相应的横切关注点以Aspect形式实现之后，会通过特定的编译器，将实现后的Aspect编译并织入到系统的静态类中。比如，AspectJ会使用ajc编译器将各个Aspect以Java字节码的形式编译到系统的各个功能模块中，以达到融合Aspect和Class的目的。而像EJB所提供的声明性事务等AOP关注点的实现，也应该归入第一代AOP行列。只不过，所采用的实现机制不同，但特点是一样的（后面将提到Java平台上实现AOP的各种机制）。

静态AOP的优点是，Aspect直接以Java字节码的形式编译到Java类中，Java虚拟机可以像通常一样加载Java类运行（因为编译完成的Aspect是完全符合Java类的规范的），不会对整个系统的运行造成任

何的性能损失。

缺点嘛，就是灵活性不够。如果横切关注点需要改变织入到系统的位置，就需要重新修改Aspect定义文件，然后使用编译器重新编译Aspect并重新织入到系统中。

7.2.2 动态 AOP 时代

动态AOP，又称为第二代AOP，该时代的AOP框架或产品，大都通过Java语言提供的各种动态特性来实现Aspect织入到当前系统的过程，如JBoss AOP、Spring AOP以及Nanning等AOP框架，都属于此列。在AspectJ融合了AspectWerkz框架之后，也引入了动态织入的行为，从而成为现在Java界唯一一个同时支持静态AOP和动态AOP特性的AOP实现产品。

第二代AOP的AOL大都采用Java语言实现，AOP的各种概念实体全部都是普通的Java类，所以很容易开发和集成。Aspect跟Class一样最终以Class身份作为系统的一等公民存在，与静态AOP最大的不同就是，AOP的织入过程在系统运行开始之后进行，而不是预先编译到系统类中，而且织入信息大都采用外部XML文件格式保存，可以在调整织入点以及织入逻辑单元的同时，不必变更系统其他模块，甚至在系统运行的时候，也可以动态更改织入逻辑。

但动态AOP在引入灵活性以及易用性的同时，也会不可避免地引入相应的性能问题。因为动态AOP的实现产品大都在类加载或者系统运行期间，采用对系统字节码进行操作的方式来完成Aspect到系统的织入，难免会造成一定的运行时性能损失。但随着JVM版本的提升，对反射以及字节码操作技术的更好支持，这样的性能损失在逐渐减少，大多数情况下，这种性能损失是可以容忍的。

7.3 Java 平台上的 AOP 实现机制

在Java平台上可以使用多种方式实现AOP。下面提到的几种方式是最经常使用的，而且也通过了相应AOP产品的验证，它们可都是帮助我们的AOP在Java平台走向现实的基石。

7.3.1 动态代理

JDK 1.3之后，引入了动态代理（Dynamic Proxy）机制，可以在运行期间，为相应的接口（Interface）动态生成对应的代理对象。所以，我们可以将横切关注点逻辑封装到动态代理的`InvocationHandler`中，然后在系统运行期间，根据横切关注点需要织入的模块位置，将横切逻辑织入到相应的代理类中。以动态代理类为载体的横切逻辑，现在当然就可以与系统其他实现模块一起工作了。

这种方式实现的唯一缺点或者说优点就是，所有需要织入横切关注点逻辑的模块类都得实现相应的接口，因为动态代理机制只针对接口有效。当然，之前也说了，毕竟动态代理是在运行期间使用反射，相对于编译后的静态类的执行，性能上可能稍逊一些。

Spring AOP默认情况下采用这种机制实现AOP机能。Nanning也是，只支持动态代理机制。

7.3.2 动态字节码增强

我们知道，Java虚拟机加载的class文件都是符合一定规范的，所以，只要交给Java虚拟机运行的文件符合Java class规范，程序的运行就没有问题。通常的class文件都是从Java源代码文件使用Javac编译器编译而成的，但只要符合Java class规范，我们也可以使用ASM或者CGLIB等Java工具库，在程序运行期间，动态构建字节码的class文件。

在这样的前提下，我们可以为需要织入横切逻辑的模块类在运行期间，通过动态字节码增强技术，为这些系统模块类生成相应的子类，而将横切逻辑加到这些子类中，让应用程序在执行期间使用的是

这些动态生成的子类，从而达到将横切逻辑织入系统的目的。

使用动态字节码增强技术，即使模块类没有实现相应的接口，我们依然可以对其进行扩展，而不用像动态代理那样受限于接口。不过，这种实现机制依然存在不足，如果需要扩展的类以及类中的实例方法等声明为`final`的话，则无法对其进行子类化的扩展。

Spring AOP在无法采用动态代理机制进行AOP功能扩展的时候，会使用CGLIB库的动态字节码增强支持来实现AOP的功能扩展。

7.3.3　Java 代码生成

实际上，如果我们从早期的J2EE开发走过来的话，或者具体点儿，如果我们接触过早期的EJB开发的话，就已经接触了这种类型的AOP实现。

- 是否记得现在依然让人念念不忘的容器内声明性事务支持？
- 是否还记得CMP类型的实体Bean只需要声明接口，而不用给出相应的接口实现类？
- 是否还记得大多数应用服务器提供商都会提供特定的EJB部署工具，以帮助我们进行EJB的部署？

事务属于跨越整个系统的一种横切关注点，所以，EJB容器提供的声明性事务支持，属于一种AOP功能模块实现。但早期EJB容器在实现这一功能的时候，大多会采用Java代码生成技术，这就是我们不需要提供CMP的接口实现类的原因，也是EJB容器提供商大多提供部署接口或者专有部署工具的原因。

EJB容器根据部署描述符文件提供的织入信息，会为相应的功能模块类根据描述符所提供的信息生成对应的Java代码，然后通过部署工具或者部署接口编译Java代码生成相应的Java类。之后，部署到EJB容器的功能模块类就可以正常工作了。

这种方式比较古老，也就早期的EJB容器使用最多，现在已经退休了。

7.3.4　自定义类加载器

所有的Java程序的class都要通过相应的类加载器（Classloader）加载到Java虚拟机之后才可以运行。默认的类加载器会读取class字节码文件，然后按照class字节码规范，解析并加载这些class文件到虚拟机运行。如果我们能够在这个class文件加载到虚拟机运行期间，将横切逻辑织入到class文件的话，是不是就完成了AOP和OOP的融合呢？

我们可以通过自定义类加载器的方式完成横切逻辑到系统的织入，自定义类加载器通过读取外部文件规定的织入规则和必要信息，在加载class文件期间就可以将横切逻辑添加到系统模块类的现有逻辑中，然后将改动后的class交给Java虚拟机运行。偷梁换柱得漂亮，不是吗？

通过类加载器，我们基本可以对大部分类以及相应的实例进行织入，功能与之前的几种方式相比当然强大很多。不过这种方式最大的问题就是类加载器本身的使用。某些应用服务器会控制整个的类加载体系，所以，在这样的场景下使用可能会造成一定的问题。

JBoss AOP和已经并入AspectJ项目的AspectWerkz框架都是采用自定义类加载器的方式实现。

7.3.5　AOL 扩展

AOL扩展是最强大、也最难掌握的一种方式，我们之前提到的AspectJ就属于这种方式。在这种方式中，AOP的各种概念在AOL中大都有一一对应的实体。我们可以使用扩展过的AOL，实现任何AOP概念实体甚至OOP概念实体，比如Aspect以及Class。所有的AOP概念在AOL中得到了最完美的表达。

采用扩展的AOL，在AOP概念的表述上颇具实力，使得AOP涉及的所有横切关注点逻辑在进行织入之前，可以自由自在地存活在自己的"国度"中。而且，具有强类型检查，基本可以对横切关注点要切入的系统运行时点有更全面的控制。而像"编译到静态类可以提升系统运行性能"，"Java虚拟机可以像加载平常类那样，加载已经织入相应逻辑的AO组件所在的类文件并运行"等特点，我们之前已经提过了。

不过，该方式强大的代价就是，你需要重新学习一门扩展了旧有的语言的AOL或者全新的AOL。建议你在看完Spring AOP框架之后，再做出你的决定，因为我们的观点一贯是K.I.S.S.（Keep It Simple Stupid）。

7.4 AOP 国家的公民

不管我们是打算实现自己的AOP框架，还是使用现有的AOP框架，在此之前，我们都需要先了解AOP涉及的相关概念。进入AOP国度而不知道这个国度中各种事物是什么的话，可会让你寸步难行的哦！

注意　因为AOP的一些术语在各个框架中可能存在差异，没有一个统一的规定，所以下面大部分概念都是以AspectJ中的概念为主线，穿插其他AOP框架中特有概念的说明，毕竟人家AspectJ出身正统嘛。

另外，AOP的一些术语本来就不统一，所以，各个概念会以英文原文阐述，而不会给出相应的中文的翻译。实际上，只要你知道这个单词代表的AOP语意就可以了，就好像你知道"Spring"代表一个Java框架，它可以帮助你做什么事情，而没有必要非要跟人说"这个春天框架怎么怎么的"一样。不过，如果你实在想知道中文怎么翻译，可以参考已出版的几本中文的Spring书，其中会提及这些名词的中文名称。不过也是一家一个说法，个人感觉乱上加乱而已。

7.4.1 Joinpoint

在系统运行之前，AOP的功能模块都需要织入到OOP的功能模块中。所以，要进行这种织入过程，我们需要知道在系统的哪些执行点上进行织入操作，这些将要在其之上进行织入操作的系统执行点就称之为Joinpoint。

为了便于理解，我们参看图7-6中的比较一般的程序流程图。

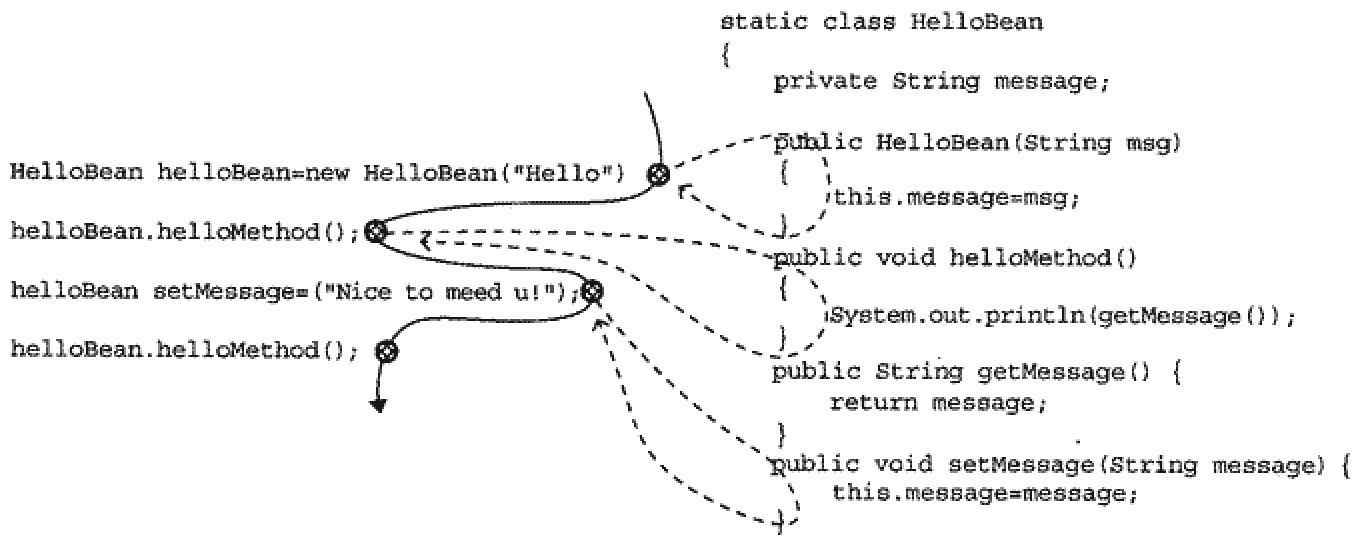

图7-6　一般程序流程图

在图7-6中，左边为一个方法的简单逻辑，实线表示的是该方法执行的顺序，虚线表示的是更加内部的调用顺序。

我们可以在HelloBean初始化的执行点进行横切逻辑的织入，可以在helloMethod方法被调用的执行点上进行横切逻辑的织入，可以在helloMethod方法内部执行的开始时点上进行织入，也可以在message字段被设置或者取得的执行点上进行横切逻辑的织入。基本上，只要允许，程序执行过程中的任何时点都可以作为横切逻辑的织入点，而所有这些执行时点都是Joinpoint。

以下是一些较为常见的Joinpoint类型。

方法调用（Method Call）。当某个方法被调用的时候所处的程序执行点，图7-6中的后面三个“圆圈”所标记的时点都属于这种类型。

方法调用执行（Method Call execution）。称之为方法执行或许更简洁，该Joinpoint类型代表的是某个方法内部执行开始时点，应该与方法调用类型的Joinpoint进行区分。我们看一下以上方法调用和执行的Sequence图或许更好理解（见图7-7）。

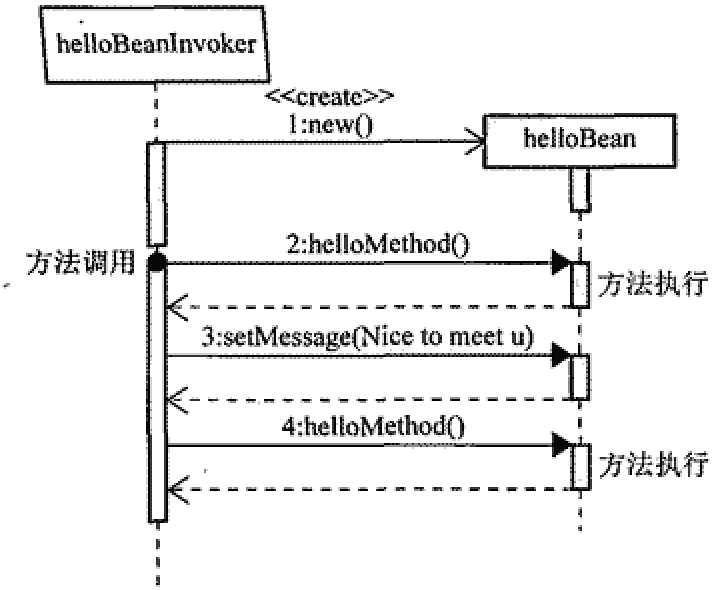

图7-7 方法调用与执行相关Sequence图

方法调用（method call）是在调用对象上的执行点，而方法执行（method execution）则是在被调用到的方法逻辑执行的时点。对于同一对象，方法调用要先于方法执行。

构造方法调用（Constructor Call）。程序执行过程中对某个对象调用其构造方法进行初始化的时点，也就是图7-6中如下代码所在的执行点。

```
HelloBean helloBean = new HelloBean("Hello!");
```

构造方法执行（Constructor Call Execution）。构造方法执行和构造方法调用之间的关系类似于方法执行和方法调用之间的关系，指的是某个对象构造方法内部执行的开始时点。

字段设置（Field Set）。对象的某个属性通过setter方法被设置或者直接被设置的时点。该Joinpoint的本质是对象的属性被设置，而通过setter方法设置还是直接设置触发的时点是相同的。

字段获取（Field Get）。相对于字段设置型的Joinpoint，字段获取型的Joinpoint，对应的是某个对象相应属性被访问的时点。可以通过getter方法访问，当然也可以直接访问。

异常处理执行（Exception Handler Execution）。该类型的Joinpoint对应程序执行过程中，在某些类型异常抛出后，对应的异常处理逻辑执行的时点。

类初始化（Class initialization）。类初始化型的Joinpoint，指的是类中某些静态类型或者静态块的初始化时点。比如，代码清单7-1中的log4j初始化位置即属于该类型Joinpoint对应位置的一种。

代码清单7-1　类初始化型Joinpoint示例

```
class FooBar
{
   static
   {
      // 初始化log4j
      BasicConfigurator.configure();
   }
   private Foo attribute;
   ...
   // 其他方法定义
   ...
}
```

基本上，程序执行过程中你认为必要的执行时点都可以作为Joinpoint。不过，对于一些位置，具体的AOP实现产品在捕捉的时候可能存在一定的困难，或者说能够实现，但付出太多却可能收效甚微。比如，程序中某个循环开始的时点也可以作为一种Joinpoint，但较难捕捉，所以，现在的AOP产品大都不支持该类型的Joinpoint。

7.4.2　Pointcut

Pointcut概念代表的是Joinpoint的表述方式。将横切逻辑织入当前系统的过程中，需要参照Pointcut规定的Joinpoint信息，才可以知道应该往系统的哪些Joinpoint上织入横切逻辑。

以图7-6中的`helloBean.helloMethod()`所在位置的Joinpoint为例，该方法在程序中两个地方被调用，而我们系统在这两个地方都要织入相应的横切逻辑，那么，我们就可以通过以下Pointcut表述来指定这两个Joinpoint：系统中`HelloBean`类的`helloMethod()`方法被调用的所有Joinpoint。

我们使用自然语言声明了一个Pointcut，该Pointcut指定了系统中符合条件的一组Joinpoint。不过，在实际系统中我们不可能使用自然语言形式的Pointcut。

1. Pointcut的表述方式

既然我们现在还没有使用自然语言编程的系统，那么就需要寻找自然语言之外的方式来表述相应的Pointcut定义。当前的AOP产品所使用的Pointcut表达形式通常可以简单划分为以下几种。

直接指定Joinpoint所在方法名称。这种形式的Pointcut表述方式比较简单，而且功能单一，通常只限于支持方法级别Joinpoint的AOP框架，或者只是方法调用类型的Joinpoint，或者只是方法执行类型的Joinpoint。而且，即使是只针对方法级别的Joinpoint，因为系统中需要织入横切逻辑的方法可能很多，一个一个地指定则过于不便，所以这种方式通常只限于Joinpoint较少且较为简单的情况。

正则表达式。这是比较普遍的Pointcut表达方式，可以充分利用正则表达式的强大功能，来归纳表述需要符合某种条件的多组Joinpoint。几乎现在大部分的Java平台的AOP产品都支持这种形式的Pointcut表达形式，包括Jboss AOP、Spring AOP以及AspectWerkz等。我们将在稍后看到Spring的AOP框架是如何使用正则表达式来指定Pointcut的。

使用特定的Pointcut表述语言。这是一种最为强大的表达Pointcut的方式，灵活性也很好，但具体实现起来可能过于复杂，需要设计该表述语言的语法，实现相应的解释器等许多工作。AspectJ使用这种方式来指定Pointcut，它提供了一种类似于正则表达式的针对Pointcut的表述语言，在表达Pointcut方

面支持比较完善。而且，Spring从发布2.0版本之后，借助于AspectJ的Pointcut表述语言解释器，现在也支持使用AspectJ的Pointcut表述语言来指定Pointcut。

2. Pointcut运算

通常，Pointcut与Pointcut之间还可以进行逻辑运算。这样，我们就可以从简单的Pointcut开始，然后通过逻辑运算，得到最终需要的可能较为复杂的Pointcut。打个比方，假设某一学校中的每个学生都算作单独的Joinpoint，那么我们可以声明以下Pointcut（见表7-1）。

表7-1 Pointcut声明示例

Pointcut声明	Pointcut声明的意思
`Pointcut.Males`	全校所有男同学
`Pointcut.Females`	全校所有女同学
`Pointcut.MalesOfGrade2Class1`	二年级一班所有男同学
`Pointcut.FemalesOfGrade2Class1`	二年级一班所有女同学

在这些Pointcut之上，我们就可以执行逻辑运算，以得到我们想要的更为确切的Pointcut表述（以Java语言中的逻辑运算符为例）。表7-2给出了几个Pointcut逻辑运算的示例。

表7-2 Pointcut逻辑运算示例

Pointcut逻辑运算	运算结果
`Pointcut.Males || Pointcut.FemalesOfGrade2Class1`	全校男生以及二年一班所有女生
`Pointcut.Males || Pointcut.Females`	全校学生

具体使用的逻辑运算语法，会因AOP产品实现的不同而不同。比如在Spring的配置文件中使用and、or等单词作为逻辑运算符，而在AspectJ中，则可以使用&&以及||。不要忘了，AspectJ可是扩展自Java的哦，所以，运算符能相同就相同了。

7.4.3 Advice

Advice是单一横切关注点逻辑的载体，它代表将会织入到Joinpoint的横切逻辑。如果将Aspect比作OOP中的Class，那么Advice就相当于Class中的Method。

按照Advice在Joinpoint位置执行时机的差异或者完成功能的不同，Advice可以分成多种具体形式。

1. Before Advice

Before Advice是在Joinpoint指定位置之前执行的Advice类型。通常，它不会中断程序执行流程，但如果必要，可以通过在Before Advice中抛出异常的方式来中断当前程序流程。如果当前Before Advice将被织入到方法执行类型的Joinpoint，那么这个Before Advice就会先于方法执行而执行。

通常，可以使用Before Advice做一些系统的初始化工作，比如设置系统初始值，获取必要系统资源等。当然，并非就限于这些情况。如果要用Before Advice来封装安全检查的逻辑，也不是不可以的，但通常情况下，我们会使用另一种形式的Advice。

2. After Advice

顾名思义，After Advice就是在相应连接点之后执行的Advice类型，但该类型的Advice还可以细分为以下三种。

- After returning Advice。只有当前Joinpoint处执行流程正常完成后，After returning Advice才会执行。比如方法执行正常返回而没有抛出异常。
- After throwing Advice。又称Throws Advice，只有在当前Joinpoint执行过程中抛出异常的情况下，才会执行。比如某个方法执行类型的Joinpoint抛出某异常而没有正常返回。
- After Advice。或许叫After (Finally) Advice更为确切，该类型Advice不管Joinpoint处执行流程是正常终了还是抛出异常都会执行，就好像Java中的`finally`块一样。

如果以方法执行类型的Joinpoint为例，我们的各种Advice的执行时机可以基本如图7-8所示。

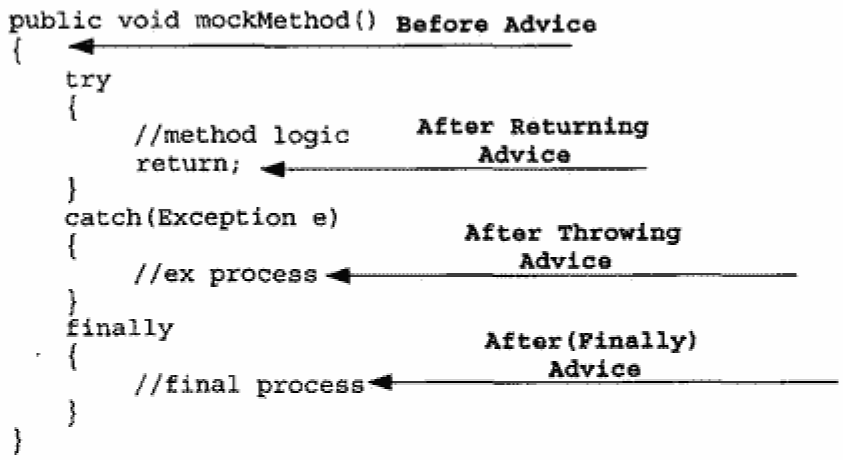

图7-8　各种Advice的执行时机

3. Around Advice

AOP Alliance属下的AOP实现大都采用拦截器（Interceptor）的叫法，但完成的功能是一样的。Around Advice对附加其上的Joinpoint进行“包裹”，可以在Joinpoint之前和之后都指定相应的逻辑，甚至于中断或者忽略Joinpoint处原来程序流程的执行。

Around Advice的行为可以因为你而发生改变。呵呵，就好像这“居心叵测”的“叵”字，中间的那一“口”就是Joinpoint，上下一横就好像是要执行的逻辑。当我们只是希望在Joinpoint之前和之后执行横切逻辑，而忽略原来Joinpoint处的逻辑执行的时候，就是居心“叵”测了。而正常情况下，Around Advice应该像一个“巨”字，我们执行完Joinpoint之前的逻辑之后，会接着走Joinpoint，然后才是Joinpoint之后的逻辑。

既然Around Advice可以在Joinpoint之前和之后都能执行相应的逻辑，那么，它自然可以完成Before Advice和After Advice的功能。不过，通常情况下，还是应该根据场景选用更为具体的Advice类型。

Around Advice应用场景非常广泛，我想大家对于J2EE中的Servlet规范提供的Filter功能应该很熟悉吧。实际上，它就是Around Advice的一种体现。使用它，我们就可以完成“资源初始化”、“安全检查”之类横切系统的关注点了。

4. Introduction

在AspectJ中称Inter-Type Declaration，在JBoss AOP中称Mix-in，都指的是这同一种类型的Advice。与之前的几种Advice类型不同，Introduction不是根据横切逻辑在Joinpoint处的执行时机来区分的，而是根据它可以完成的功能而区别于其他Advice类型。

Introduction可以为原有的对象添加新的特性或者行为，这就好像你是一个普通公民，当让你穿军装，带军帽，添加了军人类型的Introduction之后，你就拥有军人的特性或者行为。

Introduction类型的Advice因实现技术的不同，在具体软件环境中可能存在性能差异。AspectJ采用

静态织入的形式，那么对象在使用的时候，Introduction逻辑已经是编译织入完成的。所以理论上来说，AspectJ提供的Introduction类型的Advice，在现有Java平台上的AOP实现中是性能最好的；而像JBoss AOP或者Spring AOP等采用动态织入的AOP实现，Introduction的性能则要稍逊一筹。在具体的使用中，需要根据具体场景以权衡各方案之利弊。

7.4.4 Aspect

Aspect是对系统中的横切关注点逻辑进行模块化封装的AOP概念实体。通常情况下，Aspect可以包含多个Pointcut以及相关Advice定义。比如，以AspectJ形式定义的Aspect如代码清单7-2所示。

代码清单7-2 AspectJ风格定义的Aspect声明

```
aspect AjStyleAspect
{
  // pointcut定义
  pointcut query():call(public * get*(..));
  pointcut update(): execution(public void update*(..));
  ...

  // advice定义
  before(): query()
  {
    ...
  }
  after() returnint: update()
  {
    ...
  }
  ...
}
```

Spring AOP最初没有"完全"确切的实体对应真正的Aspect的概念。在2.0发布后，因为集成了AspectJ，所以可以通过使用`@AspectJ`的注解并结合普通的POJO来声明Aspect。

7.4.5 织入和织入器

毛主席有诗句云，"一桥飞架南北，天堑变通途"，织入（Weaving）过程就是"飞架"AOP和OOP的那座桥，只有经过织入过程之后，以Aspect模块化的横切关注点才会集成到OOP的现存系统中。而完成织入过程的那个"人"就称之为织入器（Weaver）啦！

AspectJ有专门的编译器来完成织入操作，即ajc，所以ajc就是AspectJ完成织入的织入器；JBoss AOP采用自定义的类加载器来完成最终织入，那么这个自定义的类加载器就是它的织入器；Spring AOP使用一组类来完成最终的织入操作，`ProxyFactory`类则是Spring AOP中最通用的织入器。总之，Java平台各AOP实现的织入器形式不一而足，唯一相同的就是它们的职责，即完成横切关注点逻辑到系统的最终织入。

7.4.6 目标对象

符合Pointcut所指定的条件，将在织入过程中被织入横切逻辑的对象，称为目标对象（Target Object）。当把所有这些概念组织到一个场景之后，我们脑海中应该有这么一幅图（见图7-9）。

其他AOP概念，我们将在稍后针对Spring AOP的讲解中附带提及，这里就不做更多描述了。

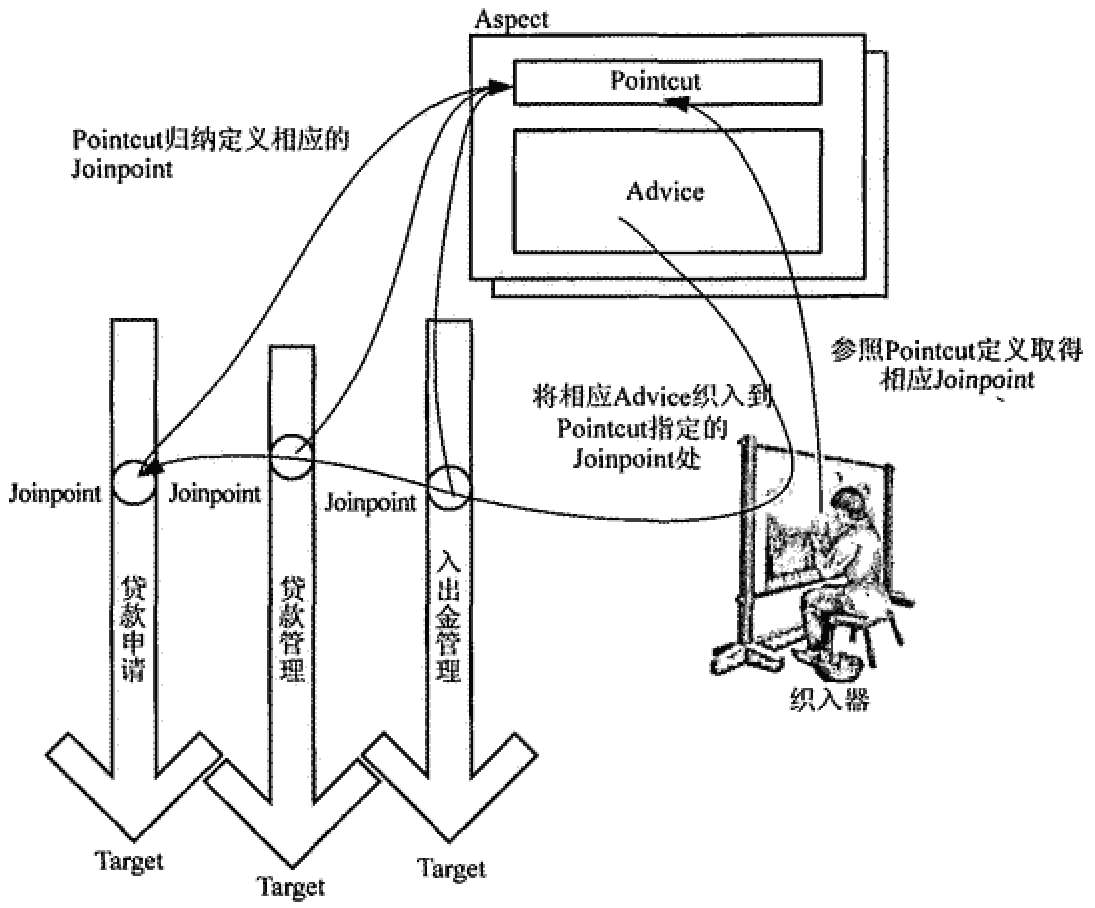

图7-9 AOP各个概念所处的场景

7.5 小结

AOP是近年流行起来的一种软件开发模式。本章伊始，我们对AOP产生的背景做了简短的介绍。之后对Java平台上各种AOP框架及产品的实现原理和方式进行了简单的剖析，以期你能够对Java平台上AOP的现状有一个总体的认识。

因为AOP涉及许多新的概念，所以在继续讲解本部分主体内容前，我们首先要对AOP王国中的各位公民有一个大致的了解。在此基础上，我们再开始真正的Spring AOP之旅。

第8章

Spring AOP概述及其实现机制

本章内容

- Spring AOP概述
- Spring AOP的实现机制

在初步领略了AOP王国的风光之后，下面该是我们踏入Spring AOP领地的时候了。

8.1 Spring AOP 概述

任何AOP的实现产品都是在构筑实业，没有这些AOP产品的支持，一切AOP的理论和概念都是空中楼阁。Spring AOP就是这个群体中的一员。

Spring AOP是Spring核心框架的重要组成部分，通常认为它与Spring的IoC容器以及Spring框架对其他JavaEE服务的集成共同组成了Spring框架的“质量三角”，足见其地位之重要（见图8-1）。

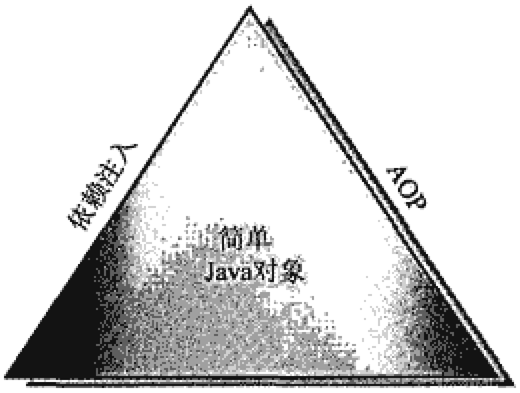

图8-1　Spring框架的“质量三角”

Spring AOP采用Java作为AOP的实现语言（AOL），较之像AspectJ那种语言扩展型的AOP实现，Spring AOP可以更快捷地融入开发过程，学习曲线相对要平滑得多。而且，Spring AOP的设计哲学也是简单而强大。它不打算将所有的AOP需求全部囊括在内，而是要以有限的20%的AOP支持，来满足80%的AOP需求。如果觉得Spring AOP无法满足你所需要的那80%之外的需求，那么求助于AspectJ好了，Spring AOP对AspectJ也提供了很好的集成。

Spring AOP的AOL语言为Java。所以，在Java语言的基础之上，Spring AOP对AOP的概念进行了适当的抽象和实现，使得每个AOP的概念都可以落到实处。而这些概念的抽象和实现，都是以我们所熟悉的Java语言的结构呈现在我们面前的。

Spring AOP从最初的框架发布以来，一直保持稳定的AOP模型。即使2.0引入了AspectJ的支持，但总的设计和实现依然延续最初的设想，也就是使用动态代理机制，构建基于Java语言的简单而强大的AOP框架。

在详细介绍Spring AOP王国的“各位公民”（AOP概念实体）之前，我们有必要先看一下这个“王国”是如何运作的。现在就开始吧！

8.2 Spring AOP 的实现机制

Spring AOP属于第二代AOP，采用动态代理机制和字节码生成技术实现。与最初的AspectJ采用编译器将横切逻辑织入目标对象不同，动态代理机制和字节码生成都是在运行期间为目标对象生成一个代理对象，而将横切逻辑织入到这个代理对象中，系统最终使用的是织入了横切逻辑的代理对象，而不是真正的目标对象。

为了理解这种差别以及最终可以达到的效果，我们有必要先从动态代理机制的根源——代理模式（Proxy Pattern）开始说起……

8.2.1 设计模式之代理模式

说到代理，我们大家应该并不陌生，现在伴随房地产的春风而繁荣的房地产中介就是一种代理。我们偶尔使用的网络代理，也是一种代理，还有许多许多，我就不一一列举了。代理处于访问者与被访问者之间，可以隔离这两者之间的直接交互，访问者与代理打交道就好像在跟被访问者在打交道一样，因为代理通常几乎会全权拥有被代理者的职能，代理能够处理的访问请求就不必要劳烦被访问者来处理了。从这个角度来说，代理可以减少被访问者的负担。另外，即使代理最终要将访问请求转发给真正的被访问者，它也可以在转发访问请求之前或者之后加入特定的逻辑，比如安全访问限制，或者，像房产中介那样抽取一定的中介费等。

在软件系统中，代理机制的实现有现成的设计模式支持，就叫代理模式。在代理模式中，通常涉及4种角色，如图8-2所示。

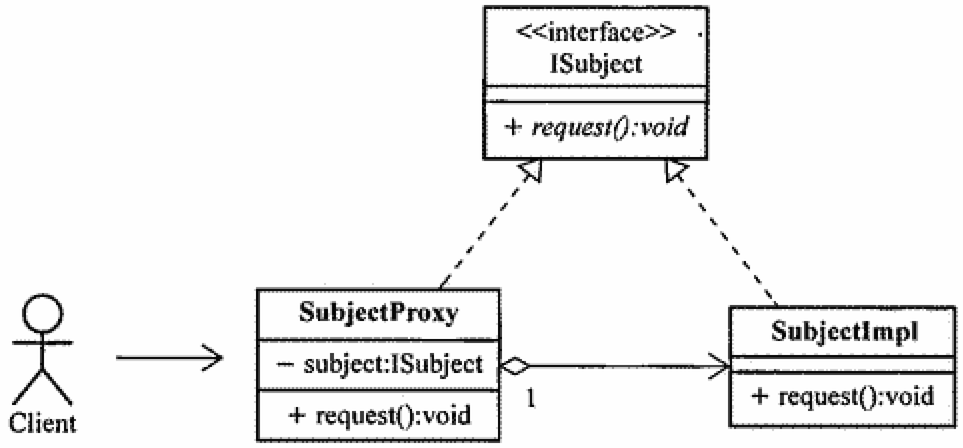

图8-2 代理模式相关类关系示意图

- ISubject。该接口是对被访问者或者被访问资源的抽象。在严格的设计模式中，这样的抽象接口是必须的，但往宽了说，某些场景下不使用类似的统一抽象接口也是可以的。
- SubjectImpl。被访问者或者被访问资源的具体实现类。如果你要访问某位明星，那么SubjectImpl就是你想访问的明星；如果你要买房子，那么SubjectImpl就是房主……
- SubjectProxy，被访问者或者被访问资源的代理实现类，该类持有一个ISubject接口的具体实例。在这个场景中，我们要对SubjectImpl进行代理，那么SubjectProxy现在持有的就是

SubjectImpl的实例。

- Client。代表访问者的抽象角色，Client将会访问ISubject类型的对象或者资源。在这个场景中，Client将会请求具体的SubjectImpl实例，但Client无法直接请求其真正要访问的资源SubjectImpl，而是必须要通过ISubject资源的访问代理类SubjectProxy进行。

SubjectImpl和SubjectProxy都实现了相同的接口ISubject，而SubjectProxy内部持有SubjectImpl的引用。当Client通过request()请求服务的时候，SubjectProxy将转发该请求给SubjectImpl。从这个角度来说，SubjectProxy反而有多此一举之嫌了。不过，SubjectProxy的作用不只局限于请求的转发，更多时候是对请求添加更多访问限制。SubjectImpl和SubjectProxy之间的调用关系，如图8-3所示。

```
public class SubjectProxy implements ISubject {
        private ISubject subject;    <---- Inject SubjectImpl to SubjectProxy
        public String request() {
                //add pre-process logic if necessary

                String originalResult=subject.request();

                // add post process logic if necessary
                return"Proxy:"+originalResult;
        }
        public ISubject getSubject() {
                return subject;
        }
        public void setSubject(ISubject subject) {
                this.subject = subject;
        }
}
```

```
public class SubjectImpl implements ISubject {
        public String request() {
                //process logic
                return"OK";
        }
}
```

图8-3 SubjectImpl和SubjectProxy之间的调用关系

在将请求转发给被代理对象SubjectImpl之前或者之后，都可以根据情况插入其他处理逻辑，比如在转发之前记录方法执行开始时间，在转发之后记录结束时间，这样就能够对SubjectImpl的request()执行的时间进行检测。或者，可以只在转发之后对SubjectImpl的request()方法返回结果进行覆盖，返回不同的值。甚至，可以不做请求转发，这样，就不会有SubjectImpl的访问发生。如果你不希望某人访问你的SubjectImpl，这种场景正好适合。

代理对象SubjectProxy就像是SubjectImpl的影子，只不过这个影子通常拥有更多的功能。如果SubjectImpl是系统中的Joinpoint所在的对象，即目标对象，那么就可以为这个目标对象创建一个代理对象，然后将横切逻辑添加到这个代理对象中。当系统使用这个代理对象运行的时候，原有逻辑的实现和横切逻辑就完全融合到一个系统中。如图8-4所示。

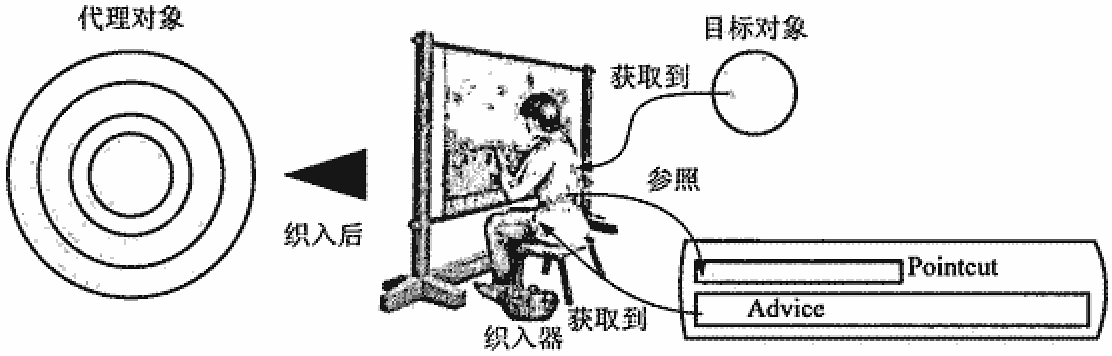

图8-4 AOP场景中的代理对象

Spring AOP本质上就是采用这种代理机制实现的，但是，具体实现细节上有所不同，让我们来看

一下为什么。

假设要对系统中所有的request()方法进行拦截，在每天午夜0点到次日6点之间，request调用不被接受，那么，我们应该为SubjectImpl提供一个ServiceControlSubjectProxy，以添加这种横切逻辑。这样就有了代码清单8-1的ServiceControlSubjectProxy定义。

代码清单8-1 ServiceControlSubjectProxy定义

```
public class ServiceControlSubjectProxy implements ISubject {
  private static final Logger = LogFactory.getLog(ServiceControlSubjectProxy.class);
  private ISubject subject;
  public LogSubjectProxy(ISubject s)
  {
    this.subject = s;
  }
  public String request() {
    TimeOfDay startTime   = new TimeOfDay(0,0,0);
    TimeOfDay endTime     = new TimeOfDay(5,59,59);
    TimeOfDay currentTime = new TimeOfDay();
    if(currentTime.isAfter(startTime) && currentTime.isBefore(endTime))
      return null;

    String originalResult = subject.request();
    return "Proxy:"+originalResult;
  }
  ...
}
```

之后，我们使用该ServiceControlSubjectProxy代替SubjectImpl使用，如以下代码所示：

```
ISubject target = new SubjectImpl();
ISubject finalSubject = new ServiceControlSubjectProxy(target);
finalSubject.request();
// 访问控制相关逻辑已经包含在请求的处理逻辑中
```

但是，系统中可不一定就ISubject的实现类有request()方法，IRequestable接口以及相应实现类可能也有request()方法，它们也是我们需要横切的关注点。IRequestable及其实现类见代码清单8-2。

代码清单8-2 IRequestable及其实现类定义

```
public interface IRequestable
{
  void request();
}

实现类

public class RequestableImpl implements IRequestable
{
  public void request()
  {
    System.out.println("request processed in RequestableImpl");
  }
}
```

为了能够为IRequestable相应实现类也织入以上的横切逻辑，我们又得提供对应的代理对象，如代码清单8-3所示。

代码清单8-3 ServiceControlRequestableProxy定义

```
public class ServiceControlRequestableProxy implements IRequestable {
  private static final Logger = LogFactory.getLog(ServiceControlRequestableProxy.class);
  private IRequestable requestable;
  public LogSubjectProxy(IRequestable requestable)
  {
    this.requestable = requestable;
  }
  public void request() {
    TimeOfDay startTime = new TimeOfDay(0,0,0);
    TimeOfDay endTime = new TimeOfDay(5,59,59);
    TimeOfDay currentTime = new TimeOfDay();
    if(currentTime.isAfter(startTime) && currentTime.isBefore(endTime))
      return;

    requestable.request();
  }
  ...
}
```

并且将该代理对象而不是目标对象绑定到系统中，如以下代码所示：

```
IRequestable target = new RequestableImpl();
IRequestable proxy  = new ServiceControlRequestableProxy(target);
proxy.request();
// 横切逻辑现在已经织入到代理对象中
```

发现问题了没有？虽然Joinpoint相同（request()方法的执行），但是对应的目标对象类型是不一样的。针对不一样的目标对象类型，我们要为其单独实现一个代理对象。而实际上，这些代理对象所要添加的横切逻辑是一样的。当系统中存在成百上千的符合Pointcut匹配条件的目标对象时（悲观点说，目标对象类型都不同），我们就要为这成百上千的目标对象创建成百上千的代理对象，不用我再往下讲了吧？这么玩会死人的！

这种为对应的目标对象创建静态代理的方法，原理上是可行的，但具体应用上存在问题，所以要寻找其他方法，以避免刚刚碰到的窘境……

8.2.2 动态代理

JDK 1.3之后引入了一种称之为动态代理（Dynamic Proxy）的机制。使用该机制，我们可以为指定的接口在系统运行期间动态地生成代理对象，从而帮助我们走出最初使用静态代理实现AOP的窘境。

动态代理机制的实现主要由一个类和一个接口组成，即java.lang.reflect.Proxy类和java.lang.reflect.InvocationHandler接口。下面，让我们看一下，如何使用动态代理来实现之前的“request服务时间控制”功能。虽然要为ISubject和IRequestable两种类型提供代理对象，但因为代理对象中要添加的横切逻辑是一样的，所以，我们只需要实现一个InvocationHandler就可以了，其定义见代码清单8-4。

代码清单8-4 RequestCtrlInvocationHandler定义

```
public class RequestCtrlInvocationHandler implements InvocationHandler {

  private static final Log logger = LogFactory.getLog(RequestCtrlInvocationHandler.class);

  private Object target;

  public RequestCtrlInvocationHandler(Object target)
```

```
	{
		this.target = target;
	}

	public Object invoke(Object proxy, Method method, Object[] args) ➥
	throws Throwable {
		if(method.getName().equals("request"))
		{
			TimeOfDay startTime = new TimeOfDay(0,0,0);
			TimeOfDay endTime = new TimeOfDay(5,59,59);
			TimeOfDay currentTime = new TimeOfDay();
			if(currentTime.isAfter(startTime) && currentTime.isBefore(endTime))
			{
				logger.warn("service is not available now.");
				return null;
			}
			return method.invoke(target, args);
		}
	return null;
	}
}
```

然后，我们就可以使用Proxy类，根据RequestCtrlInvocationHandler的逻辑，为ISubject和IRequestable两种类型生成相应的代理对象实例，见代码清单8-5。

代码清单8-5　使用Proxy和RequestCtrlInvocationHandler创建不同类型目标对象的动态代理

```
ISubject subject = (ISubject)Proxy.newProxyInstance(
	ProxyRunner.class.getClassLoader(),
	new Class[]{ISubject.class},
	new RequestCtrlInvocationHandler(new SubjectImpl()));
subject.request();

IRequestable requestable = (IRequestable)Proxy.newProxyInstance(
	ProxyRunner.class.getClassLoader(),
	new Class[]{IRequestable.class},
	new RequestCtrlInvocationHandler(new RequestableImpl()));
requestable.request();
```

即使还有更多的目标对象类型，只要它们依然织入的横切逻辑相同，用RequestCtrlInvocationHandler一个类并通过Proxy为它们生成相应的动态代理实例就可以满足要求。当Proxy动态生成的代理对象上相应的接口方法被调用时，对应的InvocationHandler就会拦截相应的方法调用，并进行相应处理。

InvocationHandler就是我们实现横切逻辑的地方，它是横切逻辑的载体，作用跟Advice是一样的。所以，在使用动态代理机制实现AOP的过程中，我们可以在InvocationHandler的基础上细化程序结构，并根据Advice的类型，分化出对应不同Advice类型的程序结构。我们将在稍后看到Spring AOP中的不同Advice类型实现以及结构规格。

动态代理虽好，但不能满足所有的需求。因为动态代理机制只能对实现了相应Interface的类使用，如果某个类没有实现任何的Interface，就无法使用动态代理机制为其生成相应的动态代理对象。虽然面向接口编程应该是提倡的做法，但不排除其他的编程实践。对于没有实现任何Interface的目标对象，我们需要寻找其他方式为其动态的生成代理对象。

默认情况下，如果Spring AOP发现目标对象实现了相应Interface，则采用动态代理机制为其生成代理对象实例。而如果目标对象没有实现任何Interface，Spring AOP会尝试使用一个称为CGLIB（Code

Generation Library）的开源的动态字节码生成类库，为目标对象生成动态的代理对象实例。

> **提示** 你可以通过参考java.lang.reflect.Proxy和java.lang.reflect.InvocationHandler的Javadoc获取更多有关动态代理的信息。*Java Reflection In Action*（Manning，2005）一书对Java的Reflection机制进行了详尽的阐述，其中有一章专门讲解了动态代理机制，不妨一读。

8.2.3 动态字节码生成

使用动态字节码生成技术扩展对象行为的原理是，我们可以对目标对象进行继承扩展，为其生成相应的子类，而子类可以通过覆写来扩展父类的行为，只要将横切逻辑的实现放到子类中，然后让系统使用扩展后的目标对象的子类，就可以达到与代理模式相同的效果了。SubClass instanceof SuperClass == true，不是吗？（图8-5演示了一个使用CGLIB进行对象行为扩展的示例。）

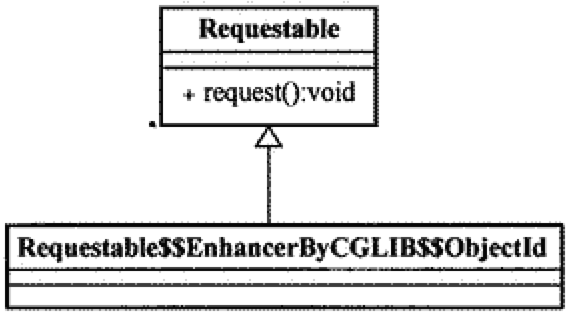

图8-5 CGLIB的继承扩展

但是，使用继承的方式来扩展对象定义，也不能像静态代理模式那样，为每个不同类型的目标对象都单独创建相应的扩展子类。所以，我们要借助于CGLIB这样的动态字节码生成库，在系统运行期间动态地为目标对象生成相应的扩展子类。

为了演示CGLIB的使用以及最终可以达到的效果，我们定义的目标类如下所示：

```
public class Requestable {
  public void request()
  {
    System.out.println("rq in Requestable " + ➥
    "without implementint any interface");
  }
}
```

CGLIB可以对实现了某种接口的类，或者没有实现任何接口的类进行扩展。但我们已经说过，可以使用动态代理机制来扩展实现了某种接口的目标类，所以，这里主要演示没有实现任何接口的目标类是如何使用CGLIB来进行扩展的。

要对Requestable类进行扩展，首先需要实现一个net.sf.cglib.proxy.Callback。不过更多的时候，我们会直接使用net.sf.cglib.proxy.MethodInterceptor接口（MethodInterceptor扩展了Callback接口）。代码清单8-6给出了针对我们的Requestable所提供的Callback实现。

代码清单8-6 RequestCtrlCallback类定义

```
public class RequestCtrlCallback implements MethodInterceptor {
  private static final Log logger = ➥
  LogFactory.getLog(RequestCtrlCallback.class);
```

```
    public Object intercept(Object object, Method method, Object[] args, ➥
    MethodProxy proxy) throws Throwable {
      if(method.getName().equals("request"))
      {
        TimeOfDay startTime = new TimeOfDay(0,0,0);
        TimeOfDay endTime = new TimeOfDay(5,59,59);
        TimeOfDay currentTime = new TimeOfDay();
        if(currentTime.isAfter(startTime) && currentTime.isBefore(endTime))
        {
          logger.warn("service is not available now.");
          return null;
        }
          return proxy.invokeSuper(object, args);
      }
      return null;
    }
}
```

这样，RequestCtrlCallback就实现了对request()方法请求进行访问控制的逻辑。现在我们要通过CGLIB的Enhancer为目标对象动态地生成一个子类，并将RequestCtrlCallback中的横切逻辑附加到该子类中，代码如下所示：

```
Enhancer enhancer = new Enhancer();
enhancer.setSuperclass(Requestable.class);
enhancer.setCallback(new RequestCtrlCallback());

Requestable proxy = (Requestable)enhancer.create();
proxy.request();
```

通过为enhancer指定需要生成的子类对应的父类，以及Callback实现，enhancer最终为我们生成了需要的代理对象实例。

使用CGLIB对类进行扩展的唯一限制就是无法对final方法进行覆写。

> **注意**　有关CGLIB的更多信息，请参照CGLIB的官方网站http://cglib.sourceforge.net/。不过，能够找到的文档不是很多，这个应该算CGLIB的缺憾吧。我觉得读其代码或许更切实际一些。

8.3　小结

在进入Spring AOP的腹地之前，我们先对Spring AOP的概况进行了介绍。接着，一起探索了Spring AOP的实现机制，包括最原始的代理模式，直至最终的动态代理与动态字节码生成。

在了解这些内容之后，我们将深入了解Spring AOP的方方面面，先从Spring AOP的最初版本说起……

第9章

Spring AOP一世

9

本章内容

- Spring AOP中的Joinpoint
- Spring AOP中的Pointcut
- Spring AOP中的Advice
- Spring AOP中的Aspect
- Spring AOP的织入
- `TargetSource`

在动态代理和CGLIB的支持下，Spring AOP框架的实现经过了两代。从Spring的AOP框架第一次发布，到Spring 2.0发布之前的AOP实现，是Spring第一代AOP实现。Spring 2.0发布后的AOP实现是第二代。不过划分归划分，Spring AOP的底层实现机制却一直没变。唯一改变的，是各种AOP概念实体的表现形式以及Spring AOP的使用方式。

下面，我们先从第一代的Spring AOP相关的概念实体说起。

9.1 Spring AOP 中的 Joinpoint

之前我们已经提到，AOP的Joinpoint可以有许多种类型，如构造方法调用、字段的设置及获取、方法调用、方法执行等。但是，在Spring AOP中，仅支持方法级别的Joinpoint。更确切地说，只支持方法执行（Method Execution）类型的Joinpoint。这一点我们从8.2节就能看出来。

虽然Spring AOP仅提供方法拦截，但是在实际的开发过程中，这已经可以满足80%的开发需求了。所以，我们不用过于担心Spring AOP的能力。

Spring AOP之所以如此，主要有以下几个原因。

(1) 前面说过了，Spring AOP要提供一个简单而强大的AOP框架，并不想因大而全使得框架本身过于臃肿。如果能够仅付出20%的努力，就能够得到80%的回报，这难道不是很好吗？*Keep It Simple, Stupid*原则指导我们抛弃旧有的EJB2时代的思想和模式，它同样适用在这里；否则，事倍功半，并不是想看到的结果。

(2) 对于类中属性（Field）级别的Joinpoint，如果提供这个级别的拦截支持，那么就破坏了面向对象的封装，而且，完全可以通过对setter和getter方法的拦截达到同样的目的。

(3) 如果应用需求非常特殊，完全超出了Spring AOP提供的那80%的需求支持，不妨求助于现有的其他AOP实现产品，如AspectJ。目前来看，AspectJ是Java平台对AOP支持最完善的产品，同时，Spring AOP也提供了对AspectJ的支持。（不过，需要注意的是，AspectJ也不是完全支持所有类型的Joinpoint，如程序中的循环结构。部分原因应该归结为要实现这20%的需求，可能需要付出80%的工作和努力。）

9.2 Spring AOP中的Pointcut

Spring中以接口定义org.springframework.aop.Pointcut作为其AOP框架中所有Pointcut的最顶层抽象，该接口定义了两个方法用来帮助捕捉系统中的相应Joinpoint，并提供了一个TruePointcut类型实例。如果Pointcut类型为TruePointcut，默认会对系统中的所有对象，以及对象上所有被支持的Joinpoint进行匹配。org.springframework.aop.Pointcut接口定义如以下代码所示：

```
public interface Pointcut {
   ClassFilter getClassFilter();
   MethodMatcher getMethodMatcher();
   Pointcut TRUE = TruePointcut.INSTANCE;
}
```

ClassFilter和MethodMatcher分别用于匹配将被执行织入操作的对象以及相应的方法。之所以将类型匹配和方法匹配分开定义，是因为可以重用不同级别的匹配定义，并且可以在不同的级别或者相同的级别上进行组合操作，或者强制让某个子类只覆写（Override）相应的方法定义等。

ClassFilter接口的作用是对Joinpoint所处的对象进行Class级别的类型匹配，其定义如下：

```
public interface ClassFilter {
   boolean matches(Class clazz);
   ClassFilter TRUE = TrueClassFilter.INSTANCE;
}
```

当织入的目标对象的Class类型与Pointcut所规定的类型相符时，matches方法将会返回true，否则，返回false，即意味着不会对这个类型的目标对象进行织入操作。比如，如果我们仅希望对系统中Foo类型的类执行织入，则可以如下这样定义ClassFilter：

```
public class FooClassFilter{
   public boolean matches(Class clazz){
      return Foo.class.equals(clazz);
   }
}
```

当然，如果类型对我们所捕捉的Joinpoint无所谓，那么Pointcut中使用的ClassFilter可以直接使用“ClassFilter TRUE = TrueClassFilter.INSTANCE;”。当Pointcut中返回的ClassFilter类型为该类型实例时，Pointcut的匹配将会针对系统中所有的目标类以及它们的实例进行。

相对于ClassFilter的简单定义，MethodMatcher则要复杂得多。毕竟，Spring主要支持的就是方法级别的拦截——“重头戏”可不能单薄啊！MethodMatcher定义如下：

```
public interface MethodMatcher {
   boolean matches(Method method, Class targetClass);
   boolean isRuntime();
   boolean matches(Method method, Class targetClass, Object[] args);
   MethodMatcher TRUE = TrueMethodMatcher.INSTANCE;
}
```

MethodMatcher通过重载（Overload），定义了两个matches方法，而这两个方法的分界线就是isRuntime()方法。在对对象具体方法进行拦截的时候，可以忽略每次方法执行的时候调用者传入的参数，也可以每次都检查这些方法调用参数，以强化拦截条件。假设对以下方法进行拦截：

```
public boolean login(String username,Sring password);
```

如果只想在login方法之前插入计数功能，那么login方法的参数对于Joinpoint捕捉就是可以忽略的。而如果想在用户登录的时候对某个用户做单独处理，如不让其登录或者给予特殊权限，那么这个方法的参数就是在匹配Joinpoint的时候必须要考虑的。

(1) 在前一种情况下，isRuntime返回false，表示不会考虑具体Joinpoint的方法参数，这种类型的MethodMatcher称之为StaticMethodMatcher。因为不用每次都检查参数，那么对于同样类型的方法匹配结果，就可以在框架内部缓存以提高性能。isRuntime方法返回false表明当前的MethodMatcher为StaticMethodMatcher的时候，只有boolean matches(Method method, Class targetClass);方法将被执行，它的匹配结果将会成为其所属的Pointcut主要依据。

(2) 当isRuntime方法返回true时，表明该MethodMatcher将会每次都对方法调用的参数进行匹配检查，这种类型的MethodMatcher称之为DynamicMethodMatcher。因为每次都要对方法参数进行检查，无法对匹配的结果进行缓存，所以，匹配效率相对于StaticMethodMatcher来说要差。而且大部分情况下，StaticMethodMatcher已经可以满足需要，最好避免使用DynamicMethodMatcher类型。如果一个MethodMatcher为DynamicMethodMatcher(isRuntime()返回true)，并且当方法boolean matches(Method method, Class targetClass);也返回true的时候，三个参数的matches方法将被执行，以进一步检查匹配条件。如果方法boolean matches(Method method,Class targetClass);返回false，那么不管这个MethodMatcher是StaticMethodMatcher还是DynamicMethodMatcher，该结果已经是最终的匹配结果——你可以猜得到，三个参数的matches方法那铁定是执行不了了。

在MethodMatcher类型的基础上，Pointcut可以分为两类，即StaticMethodMatcherPointcut和DynamicMethodMatcherPointcut。因为StaticMethodMatcherPointcut具有明显的性能优势，所以，Spring为其提供了更多支持。图9-1给出了Spring AOP中各Pointcut类型之间的一个局部“族谱”。

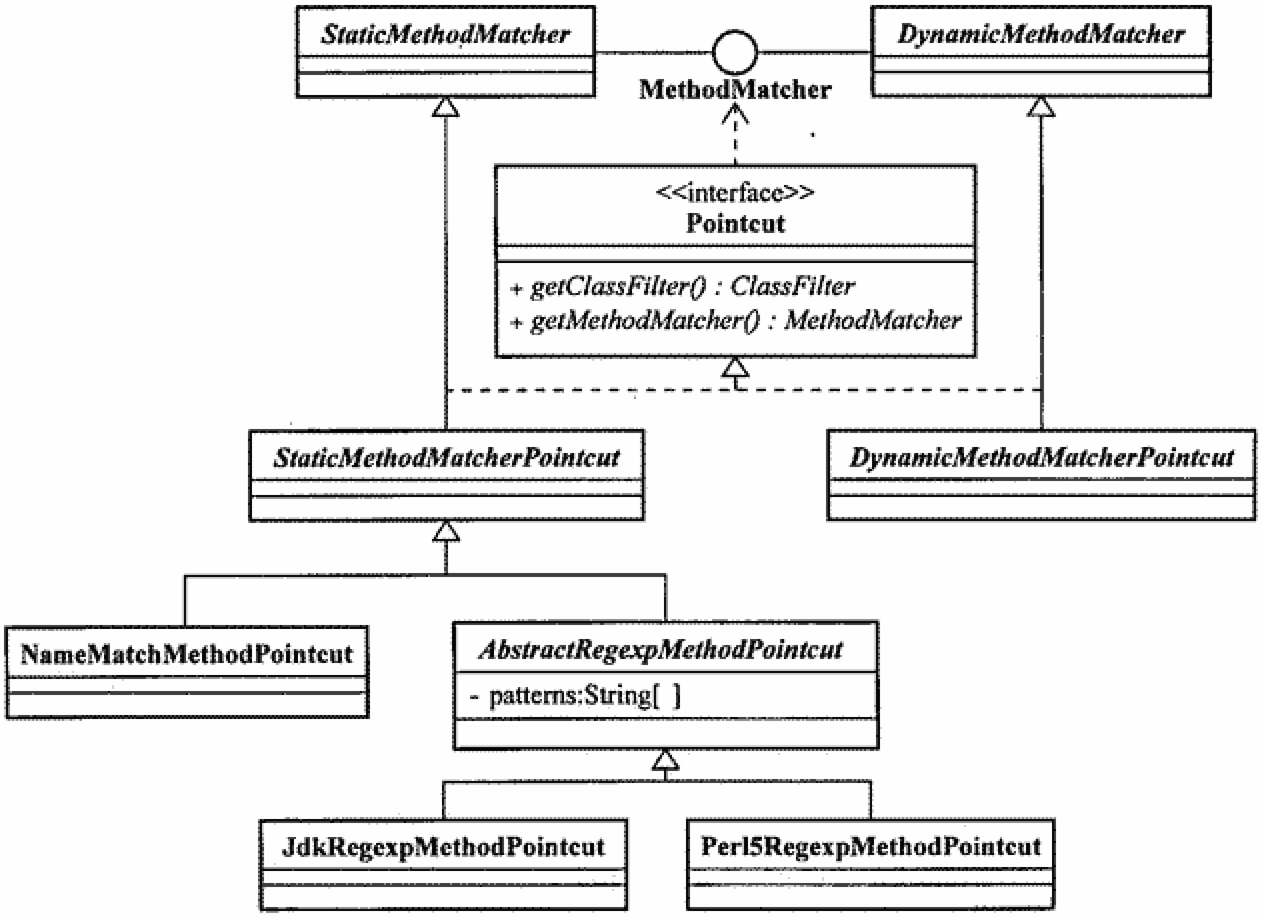

图9-1 Pointcut局部“族谱”

在深入这个“族谱”之前，我们先来看看Spring的AOP框架提供了哪些常见的Pointcut。毕竟，谁都有点儿想“坐享其成”嘛。

9.2.1 常见的 Pointcut

总地来说，图9-2给出了较为常用的几种Pointcut实现。

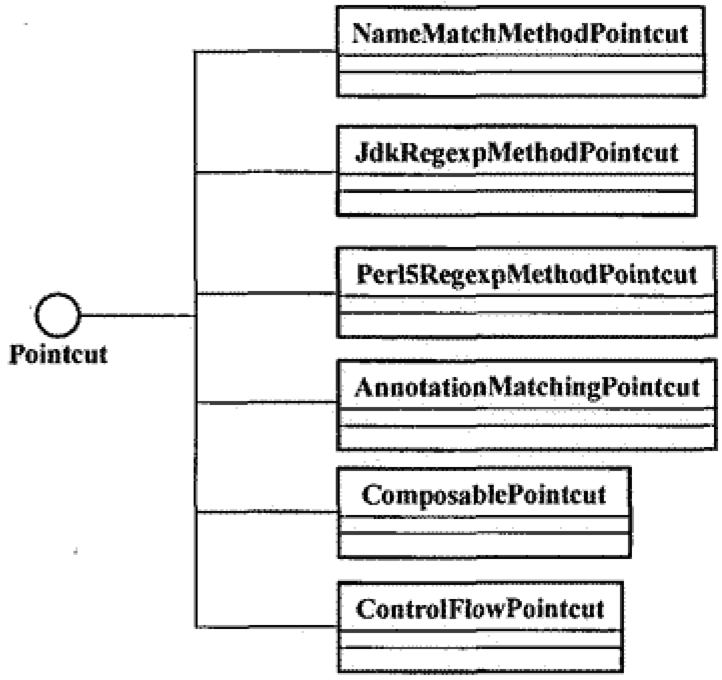

图9-2　常见的Pointcut

1. NameMatchMethodPointcut

这是最简单的Pointcut实现，属于`StaticMethodMatcherPointcut`的子类，可以根据自身指定的一组方法名称与Joinpoint处的方法的方法名称进行匹配，比如：

```
NameMatchMethodPointcut pointcut = new NameMatchMethodPointcut();
pointcut.setMappedName("matches");
// 或者传入多个方法名
pointcut.setMappedNames(new String[]{"matches","isRuntime"});
```

但是，`NameMatchMethodPointcut`无法对重载（Overload）的方法名进行匹配，因为它仅对方法名进行匹配，不会考虑参数相关信息，而且也没有提供可以指定参数匹配信息的途径。

`NameMatchMethodPointcut`除了可以指定方法名，以对指定的Joinpoint进行匹配，还可以使用“*”通配符，实现简单的模糊匹配，如下所示：

```
pointcut.setMappedNames(new String[]{"match*","*matches","mat*es"});
```

如果基于“*”通配符的`NameMatchMethodPointcut`依然无法满足对多个特定Joinpoint的匹配需要，那么使用正则表达式好了。

2. JdkRegexpMethodPointcut和Perl5RegexpMethodPointcut

`StaticMethodMatcherPointcut`的子类中有一个专门提供基于正则表达式的实现分支，以抽象类`AbstractRegexpMethodPointcut`为统帅。与`NameMatchMethodPointcut`相似，`AbstractRegexpMethodPointcut`声明了`pattern`和`patterns`属性，可以指定一个或者多个正则表达式的匹配模式（Pattern）。其下设`JdkRegexpMethodPointcut`和`Perl5RegexpMethodPointcut`两种具体实现，就如

你在图9-1中所看到的那样。

JdkRegexpMethodPointcut的实现基于JDK 1.4之后引入的JDK标准正则表达式。如果想使用该Pointcut实现，那么首先需要保证应用程序是运行在1.4或者更高版本的JVM之上。JdkRegexpMethodPointcut的简单使用示例如下：

```
JdkRegexpMethodPointcut pointcut = new JdkRegexpMethodPointcut();
pointcut.setPattern(".*match.*");
// 或者
pointcut.setPatterns(new String[]{".*match.*",".*matches"});
```

注意，使用正则表达式来匹配相应的Joinpoint所处的方法时，正则表达式的匹配模式必须以匹配整个方法签名（Method Signature）的形式指定，而不能像NameMatchMethodPointcut那样仅给出匹配的方法名称。也就是说，如果有对象定义如下：

```
package cn.spring21.sample;
public class Bar
{
  public void doSth()
  {
    ...
  }
}
```

那么，使用正则表达式.*doSth.*则会匹配Bar的doSth方法，即相当于cn.spring21.demo.Bar.doSth。但如果Pointcut使用doSth.*作为匹配的正则表达式模式，就无法捕捉到Bar的doSth方法的执行。当然，也可以通过正则表达式中的转义，更确切地指定对doSth方法的匹配：cn\.spring21\.sample\.Bar\.doSth。不过，这已经属于正则表达式如何使用的范畴了。

9

> **注意**　更多有关JDK中正则表达式的信息，可以参考1.4或者更高版本的JDK的Javadoc中java.util.regex.Pattern类定义，其中有很详细的有关正则表达式匹配模式如何指定的信息。

如果当前应用还无法使用JDK 1.4或者更高版本，或者我们更喜欢perl5风格的正则表达式，那么可以使用Perl5RegexpMethodPointcut。该Pointcut实现使用Jakarta ORO提供正则表达式支持，所以，在使用之前，请先将JakartaORO的jar包加入应用程序的classpath中。除了正则表达式的语法上可能有少许差异，Perl5RegexpMethodPointcut的使用和需要注意的问题与JdkRegexpMethodPointcut几乎相同，如下所示。

(1) 可以通过pattern或者patterns对象属性指定一个或者多个正则表达式的匹配模式。

(2) 指定的正则表达式匹配模式应该覆盖匹配整个方法签名，而不是只指定到方法名称部分。

有关该类使用的正则表达式的具体风格，请参照JakartaORO的相应文档。

3. AnnotationMatchingPointcut

不好意思，JDK又升级了。AnnotationMatchingPointcut只能用在使用JDK 5或者更高版本的应用中，因为注解是在Java 5（Tiger）发布后才有的。

AnnotationMatchingPointcut根据目标对象中是否存在指定类型的注解来匹配Joinpoint，要使用该类型的Pointcut，首先需要声明相应的注解。代码清单9-1给出了两个我们将用到的注解的定义。

代码清单9-1　用于Pointcut的ClassLevelAnnotation和MethodLevelAnnotation的定义

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
```

```
public @interface ClassLevelAnnotation {
}

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface MethodLevelAnnotation {
}
```

注解定义中的@Target指定该注解可以标注的类型。对于我们声明的两个注解来说就是，ClassLevelAnnotation用于类层次，而MethodLevelAnnotation只能用于方法层次。代码清单9-2给出了这两个注解的使用示例。

代码清单9-2 ClassLevelAnnotation和MethodLevelAnnotation使用示例

```
@ClassLevelAnnotation
public class GenericTargetObject {

  @MethodLevelAnnotation
  public void gMethod1()
  {
    System.out.println("gMethod1");
  }
  public void gMethod2()
  {
    System.out.println("gMethod2");
  }
}
```

针对代码清单9-2中的GenericTargetObject类型，不同的AnnotationMatchingPointcut定义会产生不同的匹配行为。

- AnnotationMatchingPointcut pointcut = new AnnotationMatchingPointcut(ClassLevelAnnotation.class)

AnnotationMatchingPointcut仅指定类级别的注解，GenericTargetObject类中所有方法执行的时候，将全部匹配，不管该方法指定了注解还是没有指定。也可以通过AnnotationMatchingPointcut的静态方法，来构建类级别的注解对应的AnnotationMatchingPointcut实例：

```
AnnotationMatchingPointcut pointcut = AnnotationMatchingPointcut.forClassAnnotation ➥
(ClassLevelAnnotation.class);
```

- AnnotationMatchingPointcut pointcut = AnnotationMatchingPointcut.for MethodAnnotation(MethodLevelAnnotation.class)

如果只指定方法级别的注解而忽略类级别的注解，则该Pointcut仅匹配特定的标注了指定注解的方法定义，而忽略其他方法。对于GenericTargetObject，或者其他类中任何方法来说，只要它标注了@MethodLevelAnnotation，则这些方法将都会被匹配并织入相应的横切逻辑。

- AnnotationMatchingPointcut pointcut = new AnnotationMatchingPointcut(ClassLevelAnnotation.class,MethodLevelAnnotation.class)

通过同时限定类级别的注解和方法级别的注解，我们可以进一步缩小“包围圈”。现在，只有标注了@ClassLevelAnnotation的类定义中同时标注了@MethodLevelAnnotation的方法才会被匹配，二者缺一不可。对于我们的GenericTargetObject来说，只有它的gMethod1()方法才会被拦截，而gMethod2()则不会。

注解作为外部配置方式的另一种应用配置方式（或者更高一步，Attribute-Oriented Programming），

已经愈发受到更多开发人员的青睐。从早期的xdoclet、Jakarta Commons Attributes，到Java 5引入正式的注解支持，我们现在几乎到处都能看到注解的身影——Guice、AspectJ甚至这里的AnnotationMatchingPointcut。不过，注解配置方式与外部的配置方式并不冲突，只要合理利用，二者都可以发挥更大的作用。稍后，我们就可以看到这两种方式的强大结合。

4. **ComposablePointcut**

之前在介绍Pointcut概念的时候，我们说过，Pointcut通常还提供逻辑运算功能，而ComposablePointcut就是Spring AOP提供的可以进行Pointcut逻辑运算的Pointcut实现。它可以进行Pointcut之间的“并”以及“交”运算，如：

```
ComposablePointcut pointcut1 = new ComposablePointcut(classFilter1,methodMatcher1);
ComposablePointcut pointcut2 = ...;

ComposablePointcut unitedPoincut        = pointcut1.union(pointcut2);
ComposablePointcut intersectionPointcut = pointcut1.intersection(unitedPoincut);

assertEquals(pointcut1,intersectionPointcut);
```

我们说了，Pointcut定义根据ClassFilter和MethodMatcher划分为两部分，一部分是为了重用这些定义，另一部分是为了可以相互组合。而通过ComposablePointcut，我们可以很明白地看出这两个目的，以下代码演示了这一点：

```
ComposablePointcut pointcut3 = pointcut2.union(classFilter1).intersection(methodMatcher1);
```

我们在pointcut1和pointcut3中复用了classFilter1和methodMatcher1以及pointcut2的定义，同时，还进行了Pointcut同ClassFilter以及MethodMatcher之间的逻辑组合运算。

当然，如果只想进行Pointcut与Pointcut之间的逻辑组合运算，Spring AOP提供了org.springframework.aop.support.Pointcuts工具类，其简单使用示例如下所示：

```
Pointcut pointcut1 = ...;
Pointcut pointcut2 = ...;

Pointcut unitedPoincut        = Pointcuts.union(pointcut1,pointcut2);
Pointcut intersectionPointcut = Pointcuts.intersection(pointcut1,pointcut2);
```

5. **ControlFlowPointcut**

较之其他类型的Pointcut类型，最最最特殊的Pointcut类型，ControlFlowPointcut在理解和使用上都需要我们多付出点儿脑细胞。虽然ControlFlowPointcut不是很常用，但某些场合可能需要用到，所以，还是应搞清楚这个Pointcut的特点和使用方式。

ControlFlowPointcut匹配程序的调用流程，不是对某个方法执行所在的Joinpoint处的单一特征进行匹配。

假设我们所拦截的目标对象（Target Object）有方法声明如代码清单9-3所示。

代码清单9-3 TargetObject及其调用类定义

```
public class  TargetObject
{
  public void method1()
  {
    ...
  }
  ...
}
```

```
调用类
public class TargetCaller
{
  private TargetObject target;

  public void callMethod()
  {
    target.method1();
  }
  public void setTarget(TargetObject target)
  {
    this.target = target;
  }
}
```

如果使用之前的任何Pointcut实现，我们只能指定在TargetObject的method1方法每次执行的时候，都织入相应横切逻辑。也就是说，一旦通过Pointcut指定method1处为Joinpoint，那么对该方法的执行进行拦截是必定的，不管method1是被谁调用。而通过ControlFlowPointcut，我们可以指定，只有当TargetObject的method1方法在TargetCaller类所声明的方法中被调用的时候，才对method1方法进行拦截，其他地方调用method1的话，不对method1进行拦截。如图9-3所示。

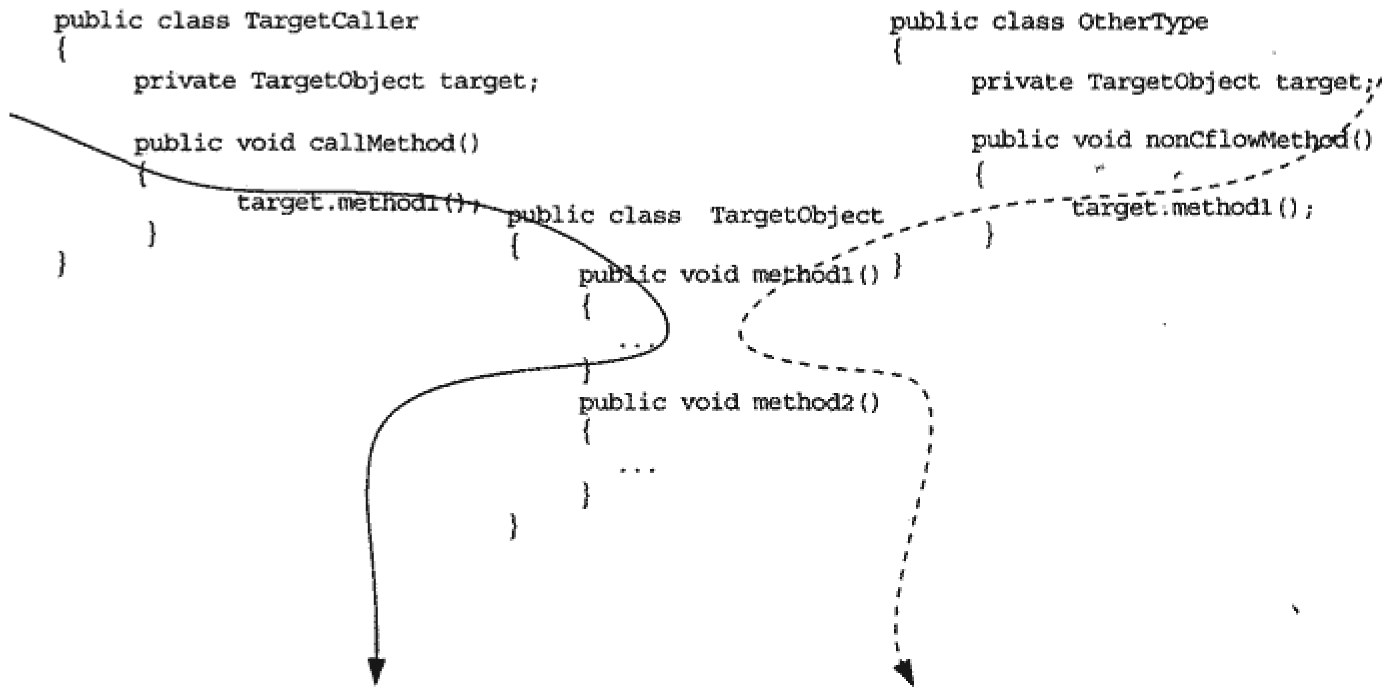

图9-3　程序的调用流程

虽然method1可以被多个对象在不同的执行流程内被调用，但是通过ControlFlowPointcut，我们可以指定，只有按照图9-3中实线所标注的执行流程，才会触发method1处的Joinpoint，而其他经过method1这个Joinpoint处的程序调用执行流程，则不会触发该Joinpoint处的横切逻辑织入操作。

使用ControlFlowPointcut，我们对以上需求的Pointcut表述，如代码清单9-4所示。

代码清单9-4　ControlFlowPointcut的使用示例

```
ControlFlowPointcut pointcut = new ControlFlowPointcut(TargetCaller.class);
Advice advice = ...;

TargetObject target = new TargetObject();
```

```
TargetObject targetObjectToUse = weaver.weave(advice).to(target).accordingto(pointcut);

// advice的逻辑在这里将不会被触发执行
targetObjectToUse.method1();

// advice的逻辑在这里将被触发执行
// 因为TargetCaller的callMethod()将调用method1
// 正像ControlFlowPointcut(TargetCaller.class)所指定的那样
TargetCaller caller = new TargetCaller();
caller.setTarget(targetObjectToUse);
caller.callMethod();
```

当织入器按照Pointcut的规定，将Advice织入到目标对象之后，从任何其他地方调用method1，是不会触发Advice所包含的横切逻辑的执行的。只有在ControlFlowPointcut规定的类内部调用目标对象的method1，才会触发Advice中横切逻辑的执行。

> **注意** 实例代码中的weaver为虚构概念，不是Spring中的织入器实现。稍后马上会为你奉上Spring AOP使用什么原理以及使用什么工具作为织入器的相关内容。

如果在ControlFlowPointcut的构造方法中单独指定Class类型的参数，那么ControlFlowPointcut将尝试匹配指定的Class中声明的所有方法，跟目标对象的Joinpoint处的方法流程组合。所以，如果只是想完成“TargetCaller.callMethod()调用TargetObject.method1()”这样的流程匹配，而忽略TargetCaller中其他方法与TargetObject中方法的Control Flow匹配，我们可以同时在构造ControlFlowPointcut的时候，传入第二个参数，即调用方法的名称：

```
ControlFlowPointcut pointcut = new ControlFlowPointcut(TargetCaller.class,"callMethod");
```

我们的TargetCaller现在就声明了一个方法（即callMethod）调用了TargetObject的方法，所以，new ControlFlowPointcut(TargetCaller.class,"callMethod");形式的声明与newControlFlowPointcut(TargetCaller.class);在当前场景下会产生相同的效果。

因为ControlFlowPointcut类型的Pointcut需要在运行期间检查程序的调用栈，而且每次方法调用都需要检查，所以性能比较差。如果不是十分必要，应该尽量避免这种Pointcut的使用。

9.2.2 扩展 Pointcut（Customize Pointcut）

虽然我认为以上Spring AOP提供的Pointcut已经足够使用，但却无法保证一定就没有更加特殊的需求，以致于以上Pointcut类型都无法满足要求。这种情况下，我们可以扩展Spring AOP的Pointcut定义，给出自定义的Pointcut实现。

要自定义Pointcut，不用白手起家，Spring AOP已经提供了相应的扩展抽象类支持，我们只需要继承相应的抽象父类，然后实现或者覆写相应方法逻辑即可。前面已经讲过，Spring AOP的Pointcut类型可以划分为StaticMethodMatcherPointcut和DynamicMethodMatcherPointcut两类。要实现自定义Pointcut，通常在这两个抽象类的基础上实现相应子类即可。

1. 自定义StaticMethodMatcherPointcut

StaticMethodMatcherPointcut根据自身语意，为其子类提供了如下几个方面的默认实现。

- 默认所有StaticMethodMatcherPointcut的子类的ClassFilter均为ClassFilter.TRUE，即忽略类的类型匹配。如果具体子类需要对目标对象的类型做进一步限制，可以通过public void setClassFilter(ClassFilter classFilter)方法设置相应的ClassFilter实现。

- 因为是StaticMethodMatcherPointcut，所以，其MethodMatcher的isRuntime方法返回false，同时三个参数的matches方法抛出UnsupportedOperationException异常，以表示该方法不应该被调用到。

最终，我们需要做的就是实现两个参数的matches方法了。

如果我们想提供一个Pointcut实现，捕捉系统里数据访问层的数据访问对象中的查询方法所在的Joinpoint，那么可以实现一个StaticMethodMatcherPointcut，如下：

```
public class QueryMethodPointcut extends StaticMethodMatcherPointcut {

  public boolean matches(Method method, Class clazz) {
    return method.getName().startsWith("get")
        && clazz.getPackage().getName().startsWith("...dao");
  }
}
```

很简单，不是吗？

> **注意**　使用现有的Pointcut类型完全可以满足相同的需求，所以在实现自定义的Pointcut之前，务必先查看一下是否已经有可用的Pointcut实现！

2. 自定义DynamicMethodMatcherPointcut

DynamicMethodMatcherPointcut也为其子类提供了部分便利。

(1) getClassFilter()方法返回ClassFilter.TRUE，如果需要对特定的目标对象类型进行限定，子类只要覆写这个方法即可。

(2) 对应的MethodMatcher的isRuntime总是返回true，同时，StaticMethodMatcherPointcut提供了两个参数的matches方法的实现，默认直接返回true。

要实现自定义DynamicMethodMatcherPointcut，通常情况下，我们只需要实现三个参数的matches方法逻辑即可。代码清单9-5给出了一个自定义的PKeySpecificQueryMethodPointcut实现类。

代码清单9-5　自定义的DynamicMethodMatcherPointcut实现示例

```
public class PKeySpecificQueryMethodPointcut extends DynamicMethodMatcherPointcut {

  public boolean matches(Method method, Class clazz, Object[] args) {
    if(method.getName().startsWith("get") ➥
    && clazz.getPackage().getName().startsWith("...dao"))
    {
      if(!ArrayUtils.isEmpty(args))
      {
        return StringUtils.equals("12345", args[0].toString());
      }
    }
    return false;
  }
}
```

如果愿意，我们也可以覆写一下两个参数的matches方法，这样，不用每次都得到三个参数的matches方法执行的时候才检查所有的条件。

9.2.3　IoC 容器中的 Pointcut

Spring中的Pointcut实现都是普通的Java对象，所以，我们同样可以通过Spring的IoC容器来注册并

使用它们。

如果某个Pointcut自身需要某种依赖，可以通过IoC容器为其注入。或者如果容器中的某个对象需要依赖于某个Pointcut，也可以把这个Pointcut注入到依赖对象中。不过，通常在使用Spring AOP的过程中，不会直接将某个Pointcut注册到容器，然后公开给容器中的对象使用。这一点稍后将详细讲述。只不过，需要说明的就是，将各个Pointcut以独立的形式注册到容器使用是完全合情合理的，如下所示：

```
<bean id="nameMatchPointcut" class="org.springframework.aop.support.NameMatchMethodPointcut">
  <property name="mappedNames">
    <list>
      <value>methodName1</value>
      <value>methodName2</value>
    </list>
  </property>
</bean>
```

9.3 Spring AOP 中的 Advice

Spring AOP加入了开源组织AOP Alliance（http://aopalliance.sourceforge.net/）[①]，目的在于标准化AOP的使用，促进各个AOP实现产品之间的可交互性。鉴于此，Spring中的Advice实现全部遵循AOP Alliance规定的接口。图9-4中就是Spring中各种Advice类型实现与AOP Alliance中标准接口之间的关系（Introduction型的Advice将单独讲解）。

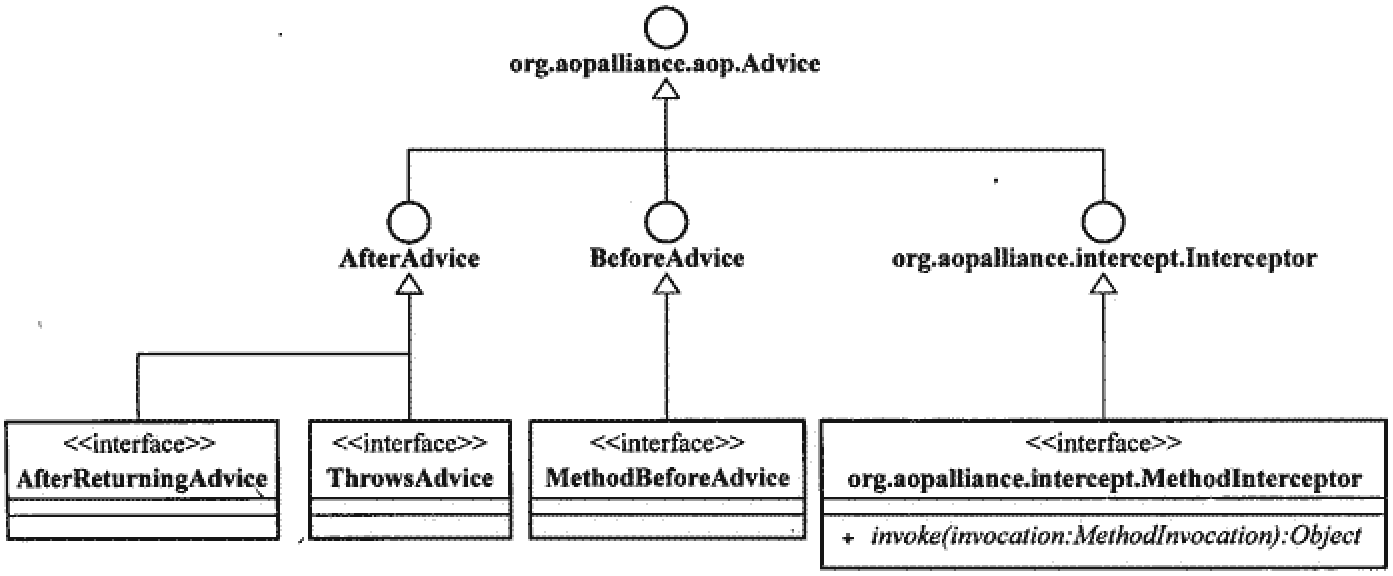

图9-4 Spring中Advice略图

Advice实现了将被织入到Pointcut规定的Joinpoint处的横切逻辑。在Spring中，Advice按照其自身实例（instance）能否在目标对象类的所有实例中共享这一标准，可以划分为两大类，即per-class类型的Advice和per-instance类型的Advice。

9.3.1 per-class 类型的 Advice

per-class类型的Advice是指，该类型的Advice的实例可以在目标对象类的所有实例之间共享。这种类型的Advice通常只是提供方法拦截的功能，不会为目标对象类保存任何状态或者添加新的特性。除

① 更确切地说，是Rod Johnson本人。

了图9-4 中没有列出的Introduction类型的Advice不属于per-class类型的Advice之外，图9-4中的所有Advice均属此列。

per-class类型的Advice将会是我们最常接触的Advice类型，所以，先从它们开刀！

1. Before Advice

本着"由简入奢"，哦，不，是"由简入深"的原则，我们先从最简单的Advice类型——Before Advice说起。

Before Advice所实现的横切逻辑将在相应的Joinpoint之前执行，在Before Advice执行完成之后，程序执行流程将从Joinpoint处继续执行，所以Before Advice通常不会打断程序的执行流程。但是如果必要，也可以通过抛出相应异常的形式中断程序流程。

要在Spring中实现Before Advice，我们通常只需要实现org.springframework.aop.MethodBeforeAdvice接口即可，该接口定义如下：

```
public interface MethodBeforeAdvice extends BeforeAdvice {
   void before(Method method, Object[] args, Object target) throws Throwable;
}
```

就像我们在图9-4中所看到的那样，MethodBeforeAdvice继承了BeforeAdvice，而BeforeAdvice与AOP Alliance的Advice一样，都是标志接口，其中没有定义任何方法。

注意　org.springframework.aop.BeforeAdvice接口没有定义方法的另一个原因在于考虑到将来的可扩展性。如果必要，可以引入支持属性级别拦截的Before Advice支持。

我们可以使用Before Advice进行整个系统的某些资源初始化或者其他一些准备性的工作。当然，其应用场景并非仅限于此，各位可以根据具体情况选择是否使用Before Advice。

假设我们的系统需要在文件系统的指定位置生成一些数据文件（系统实现中可能存在多处这样的位置），创建之前，我们需要首先检查这些指定位置是否存在，不存在则需要去创建它们。为了避免不必要的代码散落，我们可以为系统中相应目标类提供一个Before Advice，对文件系统的指定路径进行统一的检查或者初始化。代码清单9-6给出了用于初始化指定的资源路径的ResourceSetupBeforeAdvice定义。

代码清单9-6　ResourceSetupBeforeAdvice定义

```
public class ResourceSetupBeforeAdvice implements MethodBeforeAdvice {

   private Resource resource;
   public ResourceSetupBeforeAdvice(Resource resource)
   {
     this.resource = resource;
   }
   public void before(Method method, Object[] args, Object target)
        throws Throwable {
     if(!resource.exists())
     {
        FileUtils.forceMkdir(resource.getFile());
     }
   }
}
```

2. ThrowsAdvice

Spring中以接口定义org.springframework.aop.ThrowsAdvice对应通常AOP概念中的AfterThrowingAdvice。虽然该接口没有定义任何方法，但是在实现相应的ThrowsAdvice的时候，我们的方法定义需要遵循如下规则：

```
void afterThrowing([Method, args, target], ThrowableSubclass);
```

其中，[]中的三个参数可以省略。我们可以根据将要拦截的Throwable的不同类型，在同一个ThrowsAdvice中实现多个afterThrowing方法。框架将会使用Java反射机制（Java Reflection）来调用这些方法，如代码清单9-7所示。

代码清单9-7 声明了多个afterThrowing方法的ExceptionBarrierThrowsAdvice定义

```
public class ExceptionBarrierThrowsAdvice implements ThrowsAdvice {

   public void afterThrowing(Throwable t)
   {
      // 普通异常处理逻辑
   }

   public void afterThrowing(RuntimeException e)
   {
      // 运行时异常处理逻辑
   }

   public void afterThrowing(Method m,Object[] args,Object target,ApplicationException e)
   {
      // 处理应用程序生成的异常
   }
   ...
}
```

ThrowsAdvice通常用于对系统中特定的异常情况进行监控，以统一的方式对所发生的异常进行处理，我们可以在一种称之为Fault Barrier的模式中使用它。当然，我们更应该根据具体应用的场景来发挥ThrowsAdvice的最大能量。

能够马上想到的例子可能就是对系统中的运行时异常（RuntimeException）进行监控，一旦捕捉到异常，需要马上以某种方式通知系统的监控人员或者运营人员。假设我们通过email的方式发送通知，我们可以实现如代码清单9-8所示的一个ThrowsAdvice。

代码清单9-8 ExceptionBarrierThrowsAdvice的定义

```
public class ExceptionBarrierThrowsAdvice implements ThrowsAdvice {
   private JavaMailSender mailSender;
   private String[]   receiptions;

   public void afterThrowing(Method m,Object[] args,Object target,RuntimeException e)
   {
        final String exceptionMessage = ExceptionUtils.getFullStackTrace(e);
        getMailSender().send(new MimeMessagePreparator(){
        public void prepare(MimeMessage message) throws Exception {
        MimeMessageHelper helper =
           new MimeMessageHelper(message);
        helper.setSubject("...");
```

```
            helper.setTo(getReceiptions());
            helper.setText(exceptionMessage);
            }});
    }

    public JavaMailSender getMailSender() {
            return mailSender;
    }

    public void setMailSender(JavaMailSender mailSender) {
    this.mailSender = mailSender;
    }

    public String[] getReceiptions() {
            return receiptions;
    }

    public void setReceiptions(String[] receiptions) {
            this.receiptions = receiptions;
    }
}
```

该ThrowsAdvice实现中使用了Spring为JavaMail服务提供的抽象层，我们将在后面详细介绍该特性。你可以在当前ExceptionBarrierThrowsAdvice的基础上进行扩展，如添加更多配置项，或者进一步丰富邮件内容。

3. AfterReturningAdvice

org.springframework.aop.AfterReturningAdvice接口定义了Spring的AfterReturningAdvice，其定义如下：

```
public interface AfterReturningAdvice extends AfterAdvice {
   void afterReturning(Object returnValue, Method method, Object[] args, Object target) ➥
   throws Throwable;
}
```

通过Spring中的AfterReturningAdvice，我们可以访问当前Joinpoint的方法返回值、方法、方法参数以及所在的目标对象。

因为只有方法正常返回的情况下，AfterReturningAdvice才会执行，所以用来处理资源清理之类的工作并不合适。不过，如果有需要方法成功执行后进行的横切逻辑，使用AfterReturningAdvice倒比较合适。另外，虽然Spring的AfterReturningAdvice可以访问到方法的返回值，但不可以更改返回值。这一点与通常的AfterReturningAdvice的特性有所出入。好在，如果真想这么做，通过其他Advice类型还是可以做到的，如稍后即将隆重推出的Spring中的Around Advice实现。

因为Spring中的AfterReturningAdvice不能对方法返回值进行更改，所以其应用场景大受影响。不过，或多或少，还是可以寻找到某些应用的场景的。想一个更有魅力的示例有些费脑筋，所以，直接拿我遇到的场景来说吧。为了便于运营人员验证系统状态，FX的批处理程序在正常完成之后会往数据库的指定表中插入运行状态，运营人员可以通过验证这些状态判断相应的批处理任务是否成功执行，所以，我们可以实现一个AfterReturningAdvice对所有批处理任务的执行进行拦截。该AfterReturningAdvice实现见代码清单9-9。

代码清单9-9 TaskExecutionAfterReturningAdvice的定义

```
public class TaskExecutionAfterReturningAdvice implements AfterReturningAdvice {
   private SqlMapClientTemplate sqlMapClientTemplate;
```

```
    public void afterReturning(Object returnValue, Method m, Object[] args, ➥
    Object target) throws Throwable {
      Class clazz = target.getClass();
      getSqlMapClientTemplate().insert("BATCH.insertTaskStatus",clazz.getName());
    }

    public SqlMapClientTemplate getSqlMapClientTemplate() {
      return sqlMapClientTemplate;
    }

    public void setSqlMapClientTemplate(SqlMapClientTemplate sqlMapClientTemplate) {
      this.sqlMapClientTemplate = sqlMapClientTemplate;
    }
}
```

很简单，是吧！不过，如果不用AOP的话，散落到各个批处理任务中的类似逻辑可就不是两行代码可以搞定的了。

4. Around Advice

Spring AOP没有提供`After(Finally)Advice`，使得我们没有一个合适的Advice类型来承载类似于系统资源清除之类的横切逻辑。Spring AOP的`AfterReturningAdvice`不能更改Joinpoint所在方法的返回值，使得我们在方法正常返回后无法对其进行更多干预。不过，有了Around Advice，这些问题就都不是问题了。

Spring中没有直接定义对应Around Advice的实现接口，而是直接采用AOP Alliance的标准接口，即`org.aopalliance.intercept.MethodInterceptor`，该接口定义如下：

```
public interface MethodInterceptor extends Interceptor {
  Object invoke(MethodInvocation invocation) throws Throwable;
}
```

当然，从本节一开始的UML类图（图9-4）中就可以看出这个苗头了。

`MethodInterceptor`作为Around Advice那可是神通广大！之前提到的几种Advice能完成的事情，对于`MethodInterceptor`来说简直是不在话下。所以，`MethodInterceptor`，或者说Around Advice可以应用的场景那是相当地多啊！系统安全验证及检查、系统各处的性能检测、简单的日志记录[①]以及系统附加行为的添加等。

为了演示`MethodInterceptor`的使用以及需要注意的问题，我们以“简单的检测系统某些方法的执行性能”为例，实现一个`PerformanceInterceptor`，如代码清单9-10所示。

代码清单9-10 `PerformanceMethodInterceptor`的定义

```
public class PerformanceMethodInterceptor implements MethodInterceptor {
  private final Log logger = LogFactory.getLog(this.getClass());
  public Object invoke(MethodInvocation invocation) throws Throwable {
    StopWatch watch = new StopWatch();
    try
```

① 虽然大多数讲述AOP的资料或者书籍都喜欢以日志记录作为例子，但是那只是理论上的，在实际的操作过程中，实现的难度却比较大。因为系统中日志记录点很多，位置也很灵活，不可能只限于方法调用的开始或者结束等位置，即使像AspectJ这样支持很多Joinpoint类型的AOP实现产品，也无法保证能够捕捉到程序流程中的任何一个点。而且，日志记录需要的某些特定的上下文信息在取得的方式上也无定规。这两点造成日志记录无法完全的以AOP的方式进行，但简单的日志记录功能以现有的AOP产品支持还是可以的。

```
        {
          watch.start();
          return invocation.proceed();
        }
        finally
        {
          watch.stop();
          if(logger.isInfoEnabled())
          {
            logger.info(watch.toString());
          }
        }
      }
    }
```

通过MethodInterceptor的invoke方法的MethodInvocation参数，我们可以控制对相应Joinpoint的拦截行为。通过调用MethodInvocation的proceed()方法，可以让程序执行继续沿着调用链传播，这是通常我们所希望的行为。如果我们在哪一个MethodInterceptor中没有调用proceed()，那么程序的执行将会在当前MethodInterceptor处“短路”，Joinpoint上的调用链将被中断，同一Joinpoint上的其他MethodInterceptor的逻辑以及Joinpoint处的方法逻辑将不会被执行。除非你真的知道自己在做什么，否则，不要忘记调用proceed()方法哦！另外，我们还可以通过MethodInvocation对象取得Joinpoint的更多信息。

正如在PerformanceMethodInterceptor中所看到的那样，我们可以在proceed()方法，也就是Joinpoint处的逻辑执行之前或者之后插入相应的逻辑，甚至捕获proceed()方法可能抛出的异常。到现在，你是否可以理解为什么MethodInterceptor可以完成其他类型Advice可以完成的任务了呢？

为了进一步演示MethodInterceptor的使用，我们可以设想这样的场景： 如果某个销售系统规定，在商场优惠期间，所有商品一律8折出售（或者其他折扣条件），那么我们就应该在系统中所有取得商品价格的位置插入这样的横切逻辑。之前使用AfterReturningAdvice无法做的事情，现在我们使用Spring的Around Advice，也就是MethodInterceptor来做好了。代码清单9-11中声明的DiscountMethodInterceptor即用于此目的。

代码清单9-11 DiscountMethodInterceptor类的定义

```
Public class DiscountMethodInterceptor implements MethodInterceptor {
  private static final Integer  DEFAULT_DISCOUNT_RATIO = 80;
  private static final IntRange RATIO_RANGE = new IntRange(5,95);

  private Integer discountRatio = DEFAULT_DISCOUNT_RATIO;
  private boolean campaignAvailable;

  public Object invoke(MethodInvocation invocation) throws Throwable {
    Object returnValue = invocation.proceed();
    if(RATIO_RANGE.containsInteger(getDiscountRatio()) && isCampaignAvailable())
    {
      return ((Integer)returnValue)*getDiscountRatio()/100;
    }
    return returnValue;
  }

  private boolean isCampaignAvailable() {
    return campaignAvailable;
  }
```

```
    public void setCampaignAvailable(boolean campaignAvailable) {
        this.campaignAvailable = campaignAvailable;
    }

    public Integer getDiscountRatio() {
        return discountRatio;
    }

    public void setDiscountRatio(Integer discountRatio) {
        this.discountRatio = discountRatio;
    }
}
```

我们可以直接通过编程的方式来使用该类，如以下代码所示：

```
DiscountMethodInterceptor interceptor = new DiscountMethodInterceptor();
interceptor.setCampaignAvailable(true);
interceptor.setDiscountRatio(90);
...
// 现在可以将其添加到相应的Aspect中使用
```

既然我们使用了Spring框架并且这些Advice实现都是普通的POJO，更多时候，会直接将其集成到IoC容器中，如下所示：

```
<bean id="discountInterceptor" class="...DiscountMethodInterceptor">
    <property name="campaignAvailable" value="true"/>
    <property name="discountRatio" value="90"/>
</bean>
```

好啦！关于MethodInterceptor，我能说的就这些了。关于MethodInterceptor的更多应用场景，还有待你去挖掘……

9

9.3.2 per-instance 类型的 Advice

与per-class类型的Advice不同，per-instance类型的Advice不会在目标类所有对象实例之间共享，而是会为不同的实例对象保存它们各自的状态以及相关逻辑。就拿上班族为例（或许是比较痛苦的例子，呵呵），如果员工是一类人的话，那么公司的每一名员工就是员工类的不同对象实例。每个员工上班之前，公司设置了一个per-class类型的Advice进行"上班活动"的一个拦截，即打卡机，所有的员工都公用一个打卡机。当每个员工进入各自的位置之后，他们就会使用各自的电脑进行工作，而他们各自的电脑就好像per-instance类型的Advice一样，每个电脑保存了每个员工自己的资料。

在Spring AOP中，Introduction就是唯一的一种per-instance型Advice。

Introduction

Introduction可以在不改动目标类定义的情况下，为目标类添加新的属性以及行为。这就好比我们开发人员，如果公司人员紧张，没有配备测试人员，那么，通常就会给我们扣上一顶"测试人员"的帽子，让我们同时进行系统的测试工作，实际上，你还是你，只不过多了点儿事情而已。

在Spring中，为目标对象添加新的属性和行为必须声明相应的接口以及相应的实现。这样，再通过特定的拦截器将新的接口定义以及实现类中的逻辑附加到目标对象之上。之后，目标对象（确切地说是目标对象的代理对象）就拥有了新的状态和行为。这个特定的拦截器就是org.springframework.aop.IntroductionInterceptor，其定义如下：

```
public interface IntroductionInterceptor extends MethodInterceptor, ➥
DynamicIntroductionAdvice {
}
```

```
public interface DynamicIntroductionAdvice extends Advice {
   boolean implementsInterface(Class intf);
}
```

IntroductionInterceptor继承了MethodInterceptor以及DynamicIntroductionAdvice。通过DynamicIntroductionAdvice，我们可以界定当前的IntroductionInterceptor为哪些接口类提供相应的拦截功能。通过MethodInterceptor，IntroductionInterceptor就可以处理新添加的接口上的方法调用了。毕竟，原来的目标对象不会处理自己认为没有的东西啊。另外，通常情况下，对于IntroductionInterceptor来说，如果是新增加的接口上的方法调用，不必去调用MethodInterceptor的proceed()方法。毕竟，当前位置已经是"航程"的终点了（当前被拦截的方法实际上就是整个调用链中要最终执行的唯一方法）。

如果把每个目标对象实例看作盒装牛奶生产线上的那一盒盒牛奶的话，那么生产合格证就是新的Introduction逻辑，而IntroductionInterceptor就是把这些生产合格证贴到一盒盒牛奶上的那个"人"。

因为Introduction较之其他Advice有些特殊，所以，我们有必要从总体上看一下Spring中对Introduction的支持结构，图9-5给出了Introduction相关的类图结构。

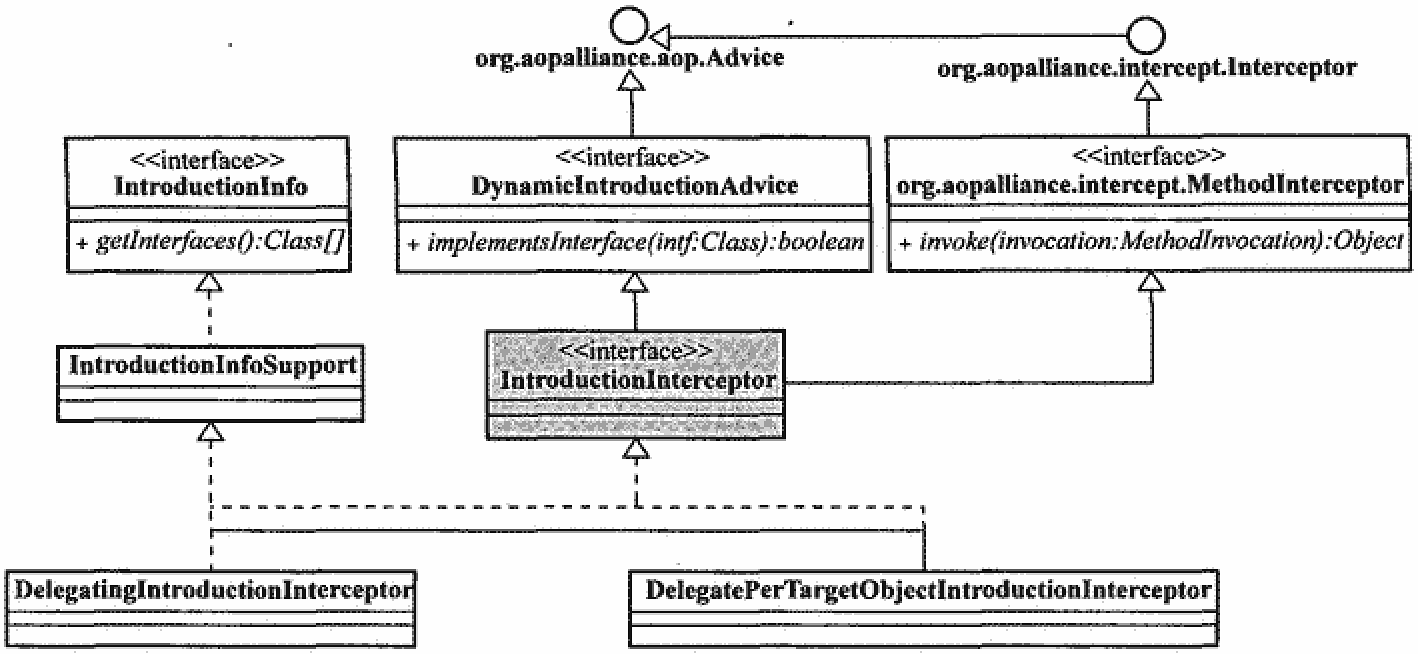

图9-5　Introduction相关类结构图

IntroductionInterceptor是从哪里出来的，前面已经讲过了，从图9-5中也可以看出相关的继承层次关系。我们要说的是实现Introduction型的Advice的两条分支，即以DynamicIntroductionAdvice为首的动态分支和以IntroductionInfo为首的静态可配置分支。从上面DynamicIntroductionAdvice的定义中可以看出，使用DynamicIntroductionAdvice，我们可以到运行时再去判定当前Introduction可应用到的目标接口类型，而不用预先就设定。而IntroductionInfo类型则完全相反，其定义如下：

```
public interface IntroductionInfo {
   Class[] getInterfaces();
}
```

实现类必须返回预定的目标接口类型，这样，在对IntroductionInfo型的Introduction进行织入

的时候，实际上就不需要指定目标接口类型了，因为它自身就带有这些必要的信息。

要对目标对象进行拦截并添加Introduction逻辑，我们可以直接扩展IntroductionInterceptor，然后在子类的invoke方法中实现所有的拦截逻辑。不过，除非特殊状况下需要去直接扩展IntroductionInterceptor，大多数时候，直接使用Spring提供的两个现成的实现类就可以了。

- DelegatingIntroductionInterceptor

从名字也可以看的出来，DelegatingIntroductionInterceptor不会自己实现将要添加到目标对象上的新的逻辑行为，而是委派（delegate）给其他实现类。不过这样也好，职责划分可以更加明确嘛！

就以简化的开发人员为例来说明DelegatingIntroductionInterceptor的用法吧！我们声明IDeveloper接口及其相关类，如代码清单9-12所示。

代码清单9-12 IDeveloper接口声明及其相关实现类定义

```
public interface IDeveloper {
  void developSoftware();
}

public class Developer implements IDeveloper {

  public void developSoftware() {
    System.out.println("I am happy with programming.");
  }
}
```

使用DelegatingIntroductionInterceptor为Developer添加新的状态或者行为，我们可以按照如下步骤进行。

(1) 为新的状态和行为定义接口。我们要为Developer添加测试人员的职能，首先需要将需要的职能以接口定义的形式声明。这样，就有了ITester声明，如下：

```
public interface ITester {
  boolean isBusyAsTester();
  void testSoftware();
}
```

(2) 给出新接口的实现类。接口实现类给出将要添加到目标对象的具体逻辑。当目标对象将要行使新的职能的时候，会通过该实现类寻求帮忙。代码清单9-13给出了针对ITester的实现类。

代码清单9-13 ITester的实现类定义

```
public class Tester implements ITester {
  private boolean busyAsTester;

  public void testSoftware() {
    System.out.println("I will ensure the quality.");
  }

  public boolean isBusyAsTester() {
    return busyAsTester;
  }

  public void setBusyAsTester(boolean busyAsTester) {
    this.busyAsTester = busyAsTester;
  }
}
```

我们可以在接口实现类中添加相应的属性甚至辅助方法，就跟实现通常的业务对象一样。

(3) 通过DelegatingIntroductionInterceptor进行Introduction的拦截。有了新增加职能的接口定义以及相应实现类，使用DelegatingIntroductionInterceptor，我们就可以把具体的Inroduction拦截委托给具体的实现类来完成，如下代码演示了这一过程：

```
ITester delegate = new Tester()
DelegatingIntroductionInterceptor interceptor = new ➥
DelegatingIntroductionInterceptor(delegate);
// 进行织入
ITester tester = (ITester)weaver.weave(developer).with(interceptor).getProxy();
tester.testSoftware();
```

(4) Introduction的最终织入过程在细节上有需要注意的地方，我们将在后面提到。虽然，DelegatingIntroductionInterceptor是Introduction型Advice的一个实现，但你可能料想不到的是，它其实是个“伪军”，因为它的实现根本就没有兑现Introduction作为per-instance型Advice的承诺。实际上，DelegatingIntroductionInterceptor会使用它所持有的同一个“delegate”接口实例，供同一目标类的所有实例共享使用。你想啊，就持有一个接口实现类的实例对象，它往哪里去放对应各个目标对象实例的状态啊？所以，如果要真的想严格达到Introduction型Advice所宣称的那样的效果，我们不能使用DelegatingIntroductionInterceptor，而是要使用它的兄弟，DelegatePerTargetObjectIntroductionInterceptor。

- DelegatePerTargetObjectIntroductionInterceptor

与DelegatingIntroductionInterceptor不同，DelegatePerTargetObjectIntroductionInterceptor会在内部持有一个目标对象与相应Introduction逻辑实现类之间的映射关系。当每个目标对象上的新定义的接口方法被调用的时候，DelegatePerTargetObjectIntroductionInterceptor会拦截这些调用，然后以目标对象实例作为键，到它持有的那个映射关系中取得对应当前目标对象实例的Introduction实现类实例。剩下的当然就是，让当前目标对象实例吃自己家锅里的饭了。如果根据当前目标对象实例没有找到对应的Introduction实现类实例，DelegatePerTargetObjectIntroductionInterceptor将会为其创建一个新的，然后添加到映射关系中。

使用DelegatePerTargetObjectIntroductionInterceptor与使用DelegatingIntroductionInterceptor没有太大的差别，唯一的区别可能就在于构造方式上。现在我们不是自己构造delegate接口实例，而只需要告知DelegatePerTargetObjectIntroductionInterceptor相应的delegate接口类型和对应实现类的类型。剩下的工作留给DelegatePerTargetObjectIntroductionInterceptor就可以了，如以下代码所示：

```
DelegatePerTargetObjectIntroductionInterceptor interceptor = ➥
new DelegatePerTargetObjectIntroductionInterceptor(DelegateImpl.class,IDelegate.class);
```

当然啦，如果DelegatingIntroductionInterceptor和DelegatePerTargetObjectIntroductionInterceptor默认的invoke方法实现逻辑无法满足你的需求，你也可以直接扩展这两个类，覆写(Override)相应的方法。不过，不知为什么，DelegatingIntroductionInterceptor和DelegatePerTargetObjectIntroductionInterceptor自身实现上对扩展有所限制，实例变量没有提供可以公开给子类的途径，一些应该声明为protected以便子类共享的方法也没有放开，而是声明为private。DelegatingIntroductionInterceptor倒是可以通过它的无参数的构造方法进行扩展，但要求子类必须同时实现新的Introduction逻辑的接口。DelegatePerTargetObjectIntroductionInterceptor干脆就没有发现什么有用的可扩展点。所以，给我的感觉就是，直接扩展这两个类跟直接扩展

IntroductionInterceptor相比，好像也没有太多优势。希望Spring Team之后能够修改这两个类，以便能够更方便地进行扩展。

要扩展IntroductionInterceptor或者DelegatingIntroductionInterceptor和Delegate-PerTargetObjectIntroductionInterceptor，通常是因为目标对象的行为，与新附加到目标对象的状态和行为相关联。这时，在处理两方面的方法调用的时候，就需要根据情况添加新的调用处理逻辑——假设Developer要进行开发的时候，检测到其作为Tester本身也在忙活，Developer要"罢工"，我们可以实现拥有类似逻辑的IntroductionInterceptor实现，如代码清单9-14所示。

代码清单9-14 扩展DelegatingIntroductionInterceptor的示例

```
public class TesterFeatureIntroductionInterceptor extends ➥
DelegatingIntroductionInterceptor implements ITester
{
  private static final long serialVersionUID = -3387097489523045796L;
  private boolean busyAsTester;

  @Override
    public Object invoke(MethodInvocation mi) throws Throwable {
      if(isBusyAsTester()
      && StringUtils.contains(mi.getMethod().getName(), "developSoftware"))
      {
        throw new RuntimeException("你想累死我呀? ");
      }
      return super.invoke(mi);
  }

  public boolean isBusyAsTester() {
    return busyAsTester;
  }

  public void setBusyAsTester(boolean busyAsTester) {
    this.busyAsTester = busyAsTester;
  }

  public void testSoftware() {
    System.out.println("I will ensure the quality.");
  }
}
```

最后要说的是Introduction的性能问题。与AspectJ直接通过编译器将Introduction织入目标对象不同，Spring AOP采用的是动态代理机制，在性能上，Introduction型的Advice要逊色不少。如果有必要，可以考虑采用AspectJ的Introduction实现。

9.4 Spring AOP 中的 Aspect

当所有的Pointcut和Advice准备好之后，就到了该把它们分门别类地装进箱子的时候了。你知道我说的箱子是什么，对吧？当然是Aspect。

在解释Aspect的概念的时候曾经提到过，Spring中最初没有完全明确的Aspect的概念，但是，这并不意味着就没有。只不过，Spring中的这个Aspect在实现和特性上有所特殊而已。

Advisor代表Spring中的Aspect，但是，与正常的Aspect不同，Advisor通常只持有一个Pointcut和一个Advice。而理论上，Aspect定义中可以有多个Pointcut和多个Advice，所以，我们可以认为Advisor是一种特殊的Aspect。

为了能够更清楚Advisor的实现结构体系，我们可以将Advisor简单划分为两个分支，一个分支以org.springframework.aop.PointcutAdvisor为首，另一个分支则以org.springframework.aop.IntroductionAdvisor为头儿，如图9-6所示。

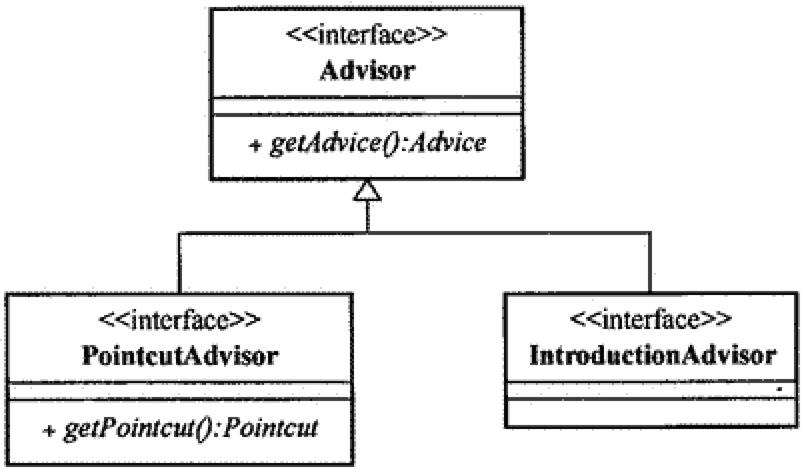

图9-6　Advisor分支

9.4.1　PointcutAdvisor 家族

实际上，org.springframework.aop.PointcutAdvisor才是真正的定义一个Pointcut和一个Advice的Advisor，大部分的Advisor实现全都是PointcutAdvisor的"部下"（见图9-7）。

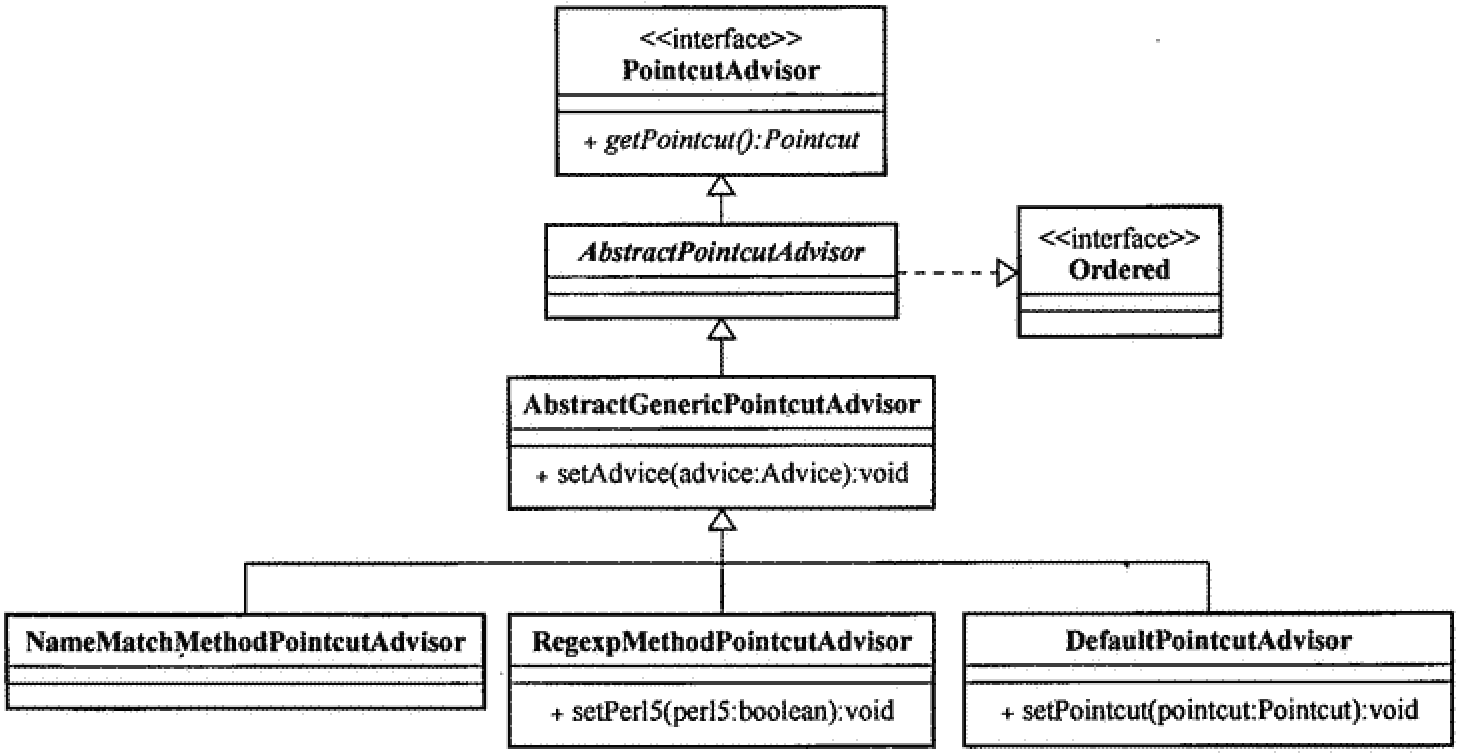

图9-7　PointcutAdvisor及相关子类

下面我们就来看一下几个常用的PointcutAdvisor实现。

1. DefaultPointcutAdvisor

DefaultPointcutAdvisor是PointcutAdvisor的"大弟子"，是最通用的PointcutAdvisor实现。除了不能为其指定Introduction类型的Advice之外，剩下的任何类型的Pointcut、任何类型的Advice

都可以通过DefaultPointcutAdvisor来使用。我们可以在构造DefaultPointcutAdvisor的时候，就明确指定属于当前DefaultPointcutAdvisor实例的Pointcut和Advice，也可以在DefaultPointcut-Advisor实例构造完成后，再通过setPointcut以及setAdvice方法设置相应的Pointcut和Advice（使用示例见代码清单9-15）。

代码清单9-15 DefaultPointcutAdvisor使用示例

```
Pointcut pointcut = ...;     // 任何Pointcut类型
Advice   advice   = ...;     // 除Introduction类型外的任何Advice类型

DefaultPointcutAdvisor advisor = new DefaultPointcutAdvisor(pointcut,advice);
// 或者
DefaultPointcutAdvisor advisor = new DefaultPointcutAdvisor(advice);
advisor.setPointcut(pointcut);
// 或者
DefaultPointcutAdvisor advisor = new DefaultPointcutAdvisor();
advisor.setPointcut(pointcut);
advisor.setAdvice(advice);
```

此处给出代码并不是让你在实际的环境中就这么用，而是为了演示事实的真相。实际上，Spring中任何的bean都可以通过IoC容器来管理，Spring AOP中的任何概念对此也同样适用。大多数时候，我们会通过IoC容器来注册和使用Spring AOP的各种概念实体。

通常使用Spring的IoC容器注册管理DefaultPointcutAdvisor的情形，如代码清单9-16所示。

代码清单9-16 通过IoC容器注册管理DefaultPointcutAdvisor相关类

```
<bean id="pointcut" class="...">
  ...
</bean>
<bean id="advice" class="...">
  ...
</bean>

<bean id="advisor" class="org.springframework.aop.support.DefaultPointcutAdvisor">
  <property name="pointcut" ref="pointcut"/>
  <property name="advice" ref="advice"/>
</bean>
```

2. NameMatchMethodPointcutAdvisor

NameMatchMethodPointcutAdvisor是细化后的DefaultPointcutAdvisor，它限定了自身可以使用的Pointcut类型为NameMatchMethodPointcut，并且外部不可更改。不过，对于使用的Advice来说，除了Introduction，其他任何类型的Advice都可以使用。

NameMatchMethodPointcutAdvisor内部持有一个NameMatchMethodPointcut类型的Pointcut实例。当通过NameMatchMethodPointcutAdvisor公开的setMappedName和setMappedNames方法设置将被拦截的方法名称的时候，实际上是在操作NameMatchMethodPointcutAdvisor所持有的这个NameMatchMethodPointcut实例。

NameMatchMethodPointcutAdvisor的使用也很简单，通过编程方式还是通过IoC容器都可以，编程方式使用示例如下：

```
Advice advice = ...; // 任何类型的Advice，Introduction类型除外
NameMatchMethodPointcutAdvisor advisor = new NameMatchMethodPointcutAdvisor(advice);
advisor.setMappedName("methodName2Intercept");
```

```
// 或者
NameMatchMethodPointcutAdvisor advisor = new NameMatchMethodPointcutAdvisor(advice);
advisor.setMappedNames(new String[]{"method1","method2"});
```

通过IoC容器使用的情形，见代码清单9-17。

代码清单9-17　通过IoC容器配置使用NameMatchMethodPointcutAdvisor

```
<bean id="advice" class="...">
  ...
</bean>

<bean id="advisor" class="org.springframework.aop.support. ➥
NameMatchMethodPointcutAdvisor">
  <property name="advice">
    <ref bean="advice"/>
  </property>
  <property name="mappedNames">
    <list>
      <value>method1</value>
      ...
    </list>
  </property>
</bean>
```

3. RegexpMethodPointcutAdvisor

与NameMatchMethodPointcutAdvisor类似，RegexpMethodPointcutAdvisor也限定了自身可以使用的Pointcut的类型，即只能通过正则表达式为其设置相应的Pointcut。

RegexpMethodPointcutAdvisor自身内部持有一个AbstractRegexpMethodPointcut的实例。希望你还记得，AbstractRegexpMethodPointcut有两个实现类，即Perl5RegexpMethodPointcut和JdkRegexpMethodPointcut。默认情况下，RegexpMethodPointcutAdvisor会使用JdkRegexpMethodPointcut。如果要强制使用Perl5RegexpMethodPointcut，那么可以通过RegexpMethodPointcutAdvisor的setPerl5(boolean)达成所愿。

RegexpMethodPointcutAdvisor提供了许多构造方法，我们可以在构造时就指定Pointcut的正则表达式匹配模式以及相应的Advice，也可以构造完成之后再指定，在使用上与其他的Advisor实现并无太多差别。我们这里只演示在IoC容器中的配置使用方式（见代码清单9-18）。

代码清单9-18　通过IoC容器配置使用RegexpMethodPointcutAdvisor示例

```
<bean id="advice" class="...">
  ...
</bean>
<bean id="advisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
  <property name="pattern">
    <value>cn\.spring21\..*\.methodNamePattern</value>
  </property>
  <property name="advice">
    <ref bean="advice"/>
  </property>
  <property name="perl5">
    <value>false</value>
  </property>
</bean>
```

关于正则表达式的更多信息可以参照JDK 1.4或者更高版本的Javadoc以及Jakarta ORO项目文档。

4. DefaultBeanFactoryPointcutAdvisor

DefaultBeanFactoryPointcutAdvisor是使用比较少的一个Advisor实现，因为自身绑定到了BeanFactory，所以，要使用DefaultBeanFactoryPointcutAdvisor，我们的应用铁定要绑定到Spring的IoC容器了。而且，通常情况下，DefaultPointcutAdvisor已经完全可以满足需求。

DefaultBeanFactoryPointcutAdvisor的作用是，我们可以通过容器中的Advice注册的beanName来关联对应的Advice。只有当对应的Pointcut匹配成功之后，才去实例化对应的Advice，减少了容器启动初期Advisor和Advice之间的耦合性。

要使用DefaultBeanFactoryPointcutAdvisor，我们通常需要在容器的配置文件中进行如代码清单9-19所示的配置。

代码清单9-19 DefaultBeanFactoryPointcutAdvisor使用示例

```
<bean id="advice" class="...">
</bean>

<bean id="pointcut" class="org.springframework.aop.support.NameMatchMethodPointcut">
  <property name="mappedName" value="doSth"/>
</bean>

<bean id="advisor" class="org.springframework.aop.support. ➥
DefaultBeanFactoryPointcutAdvisor">
  <property name="pointcut" ref="pointcut"/>
  <property name="adviceBeanName" value="advice"/>
</bean>
```

注意，对应advice的配置属性名称为“adviceBeanName”，而它的值就对应advice的beanName。除了这一点，与DefaultPointcutAdvisor使用并无二致。

9.4.2 IntroductionAdvisor 分支

IntroductionAdvisor与PointcutAdvisor最本质上的区别就是，IntroductionAdvisor只能应用于类级别的拦截，只能使用Introduction型的Advice，而不能像PointcutAdvisor那样，可以使用任何类型的Pointcut，以及差不多任何类型的Advice。也就是说，IntroductionAdvisor纯粹就是为Introduction而生的。

IntroductionAdvisor的类层次比较简单，只有一个默认实现DefaultIntroductionAdvisor①，其继承层次见图9-8。

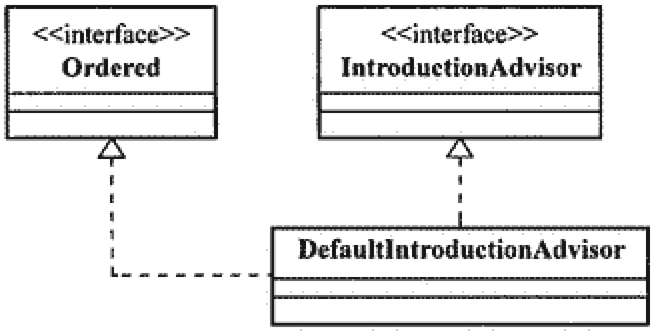

图9-8 IntroductionAdvisor类结构图

既然IntroductionAdvisor仅限于Introduction的使用场景，那么DefaultIntroductionAdvisor

---

① 另一个通过AspectJ继承扩展的实现，将在第10章介绍。

的使用也比较简单，只可以指定Introduction型的Advice（即IntroductionInterceptor）以及将被拦截的接口类型。其使用示例见代码清单9-20。

代码清单9-20　DefaultIntroductionAdvisor使用示例

```
<bean id="introductionInterceptor" class="org.springframework.aop.support. ➥
DelegatingIntroductionInterceptor">
  <constructor-arg>
    <bean class="...DelegateImpl">
    </bean>
  </constructor-arg>
</bean>

<bean id="introductionAdvisor" class="org.springframework.aop.support. ➥
DefaultIntroductionAdvisor">
  <constructor-arg><ref bean="introductionInterceptor"/></constructor-arg>
  <constructor-arg><value>...IDelegateInterface</value></constructor-arg>
</bean>
```

我们也可以指定Advice以及一个IntroductionInfo对象类来构造DefaultIntroductionAdvisor，因为IntroductionInfo可以提供必要的目标接口类型。代码清单9-21是结合IntroductionInfo使用的DefaultIntroductionAdvisor的使用示例。

代码清单9-21　结合IntroductionInfo使用的DefaultIntroductionAdvisor的使用示例

```
<bean id="introductionInterceptor" class="org.springframework.aop.support. ➥
DelegatingIntroductionInterceptor">
  <constructor-arg>
    <bean class="...DelegateImpl">
    </bean>
  </constructor-arg>
</bean>

<bean id="introductionAdvisor" class="org.springframework.aop.support. ➥
DefaultIntroductionAdvisor">
  <constructor-arg index="0"><ref bean="introductionInterceptor"/></constructor-arg>
  <constructor-arg index="1"><ref bean="introductionInterceptor"/></constructor-arg>
</bean>
```

不用奇怪为什么我们在构造DefaultIntroductionAdvisor的时候传入两个“introducetion-Interceptor”，它们两个其实是不一样的，前者是作为Introduction型的Advice实例，后者则是作为IntroductionInfo的实例。不要忘了DelegatingIntroductionInterceptor实现了Introduciton-Info接口哦！

9.4.3　Ordered 的作用

系统中只存在单一的横切关注点的情况很少，大多数时候，都会有多个横切关注点需要处理，那么，系统实现中就会有多个Advisor存在。当其中的某些Advisor的Pointcut匹配了同一个Joinpoint的时候，就会在这同一个Joinpoint处执行多个Advice的横切逻辑。如果这些Advisor所关联的Advice之间没有很强的优先级依赖关系，那么谁先执行，谁后执行都不会造成任何影响。而一旦这几个需要在同一Joinpoint处执行的Advice逻辑存在优先顺序依赖的话，就需要我们来干预了，否则，系统的行为就会偏离我们的预想。

记得有一天，同事大鹏突然问我，说：“头儿，这个任务初始化的时候抛出异常但没被我们的ThrowsAdvice截获，帮看一下呗？”我心里纳闷，不能吧？查了一下Pointcut的正则表达式定义，没错

啊，应该能捕获到啊。最后查到抛出异常的是应用到同一个方法的Advice所抛出的，我才猛然醒悟……

现在假设有两个Advisor，一个进行权限检查，当检查到当前调用没有权限的时候，抛出相应异常，称为PermissionAuthAdvisor；另一个Advisor使用一个ThrowsAdvice对系统中的所有需要检测的异常进行拦截，称其为ExceptionBarrierAdvisor。如果以如下形式声明这两个Advisor，就不会有问题：

```
<bean id="exceptionBarrierAdvisor" class="...ExceptionBarrierAdvisor">
  ...
</bean>

<bean id="pemissionAuthAdvisor" class="...PermissionAuthAdvisor">
  ...
</bean>
```

即使PermissionAuthAdvisor的Advice抛出异常，我们的ExceptionBarrierAdvisor也可以捕获该异常并进行系统内的统一处理。而如果我们像如下这样，颠倒它们两个的声明顺序，那就有问题了：

```
<bean id="pemissionAuthAdvisor" class="...PermissionAuthAdvisor">
  ...
</bean>

<bean id="exceptionBarrierAdvisor" class="...ExceptionBarrierAdvisor">
  ...
</bean>
```

在PermissionAuthAdvisor中的Advice抛出异常之后，ExceptionBarrierAdvisor并没有起作用，问题出在哪儿呢？

Spring在处理同一Joinpoint处的多个Advisor的时候，实际上会按照指定的顺序和优先级来执行它们，顺序号决定优先级，顺序号越小，优先级越高，优先级排在前面的，将被优先执行。我们可以从0或者1开始指定，因为小于0的顺序号原则上由Spring AOP框架内部使用。默认情况下，如果我们不明确指定各个Advisor的执行顺序，那么Spring会按照它们的声明顺序来应用它们，最先声明的顺序号最小但优先级最大，其次次之。

有了这些前提，我们就可以知道为什么仅颠倒两个Advisor的顺序就会造成某个Advisor失效。让我们来看图9-9。

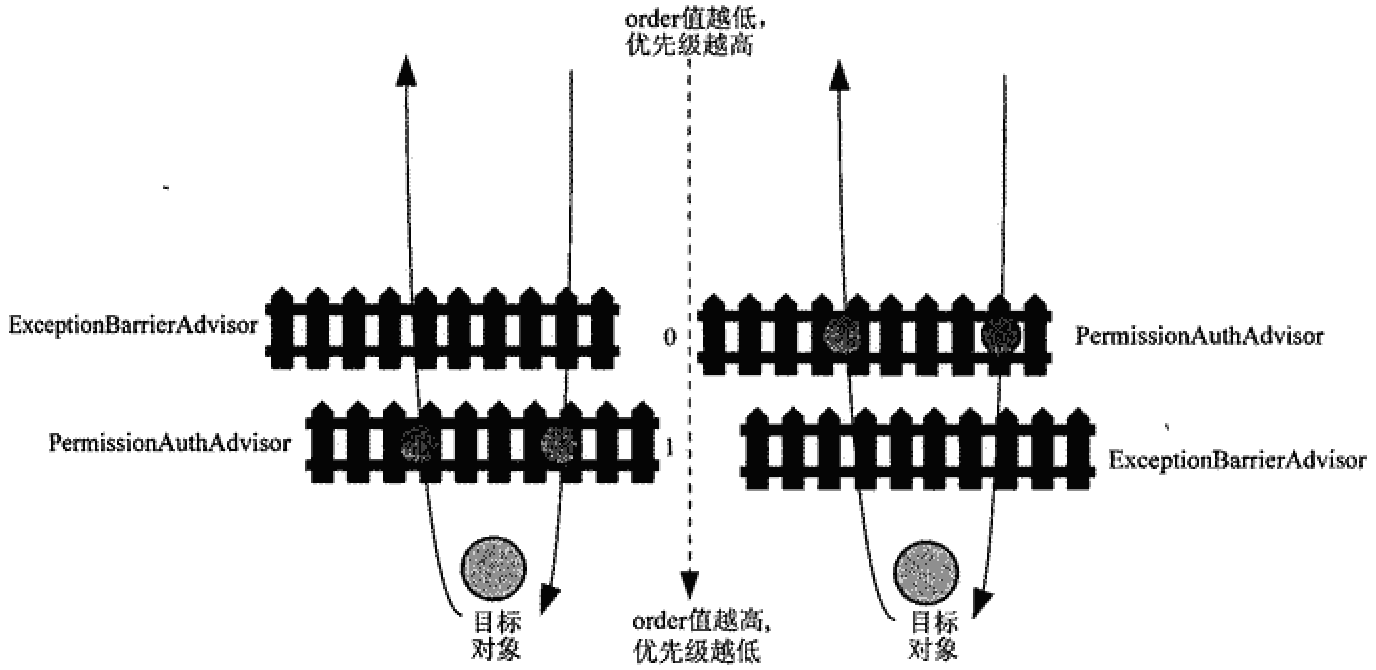

图9-9 Ordered作用示意图

在图9-9中，左边是正常的情况，当调用流程在PermissionAuthAdvisor中出现问题时，甚至是在目标对象上出现问题时，ExceptionBarrierAdvisor会在调用流程返回的时候捕获到相应异常；而右边就是调换顺序后的结果，可以看到现在PermissionAuthAdvisor上出现问题的话，因为调用流程已经经过了ExceptionBarrierAdvisor，所以，ExceptionBarrierAdvisor根本无法捕获PermissionAuthAdvisor上的异常（虽然目标对象上的问题可以捕获到）。

虽然我们可以通过调整配置中各个Advisor声明的顺序来避免以上问题，但是，这并非最彻底的解决方法。最彻底的方法就是，为每个Advisor明确指定顺序号。在Spring框架中，我们可以通过让相应的Advisor以及其他顺序紧要的bean实现org.springframework.core.Ordered接口来明确指定相应顺序号。不过，从图9-7中也应该看到了，各个Advisor实现类，其实已经实现了Ordered接口。我们无需自己去实现这个接口了，唯一要做的是直接在配置的时候指定顺序号。代码清单9-22中的配置为我们的两个Advisor指定了明确的顺序号，从而避免了最初问题的出现。

代码清单9-22　明确指定各个Advisor的顺序号的演示

```
<bean id="pemissionAuthAdvisor" class="...PermissionAuthAdvisor">
   <property name="order" value="1"/>
   ...
</bean>

<bean id="exceptionBarrierAdvisor" class="...ExceptionBarrierAdvisor">
   <property name="order" value="0"/>
   ...
</bean>
```

9.5　Spring AOP 的织入

俗话说得好，“万事俱备，只欠东风”！各个模块我们已经实现好了，剩下的工作，就是拼装各个模块。

要进行织入，AspectJ采用ajc编译器作为它的织入器；JBoss AOP使用自定义的ClassLoader作为它的织入器；而在Spring AOP中，使用类org.springframework.aop.framework.ProxyFactory作为织入器。

9.5.1　如何与 ProxyFactory 打交道

首先需要声明的是，ProxyFactory并非Spring AOP中唯一可用的织入器，而是最基本的一个织入器实现，所以，我们就从最基本的这个织入器开始，来窥探一下Spring AOP的织入过程到底是一个什么样子。

使用ProxyFactory来进行横切逻辑的织入很简单。我们知道，Spring AOP是基于代理模式的AOP实现，织入过程完成后，会返回织入了横切逻辑的目标对象的代理对象。为ProxyFactory提供必要的“生产原材料”之后，ProxyFactory就会返回那个织入完成的代理对象（如以下代码所示）：

```
ProxyFactory weaver = new ProxyFactory(yourTargetObject);
// 或者
// ProxyFactory weaver = new ProxyFactory();
// weaver.setTarget(task);
Advisor advisor = ...;
weaver.addAdvisor(advisor);
Object proxyObject = weaver.getProxy();
// 现在可以使用proxyObject了
```

使用ProxyFactory只需要指定如下两个最基本的东西。

- 第一个是要对其进行织入的目标对象。我们可以通过ProxyFactory的构造方法直接传入，也可以在ProxyFactory构造完成之后，通过相应的setter方法进行设置。
- 第二个是将要应用到目标对象的Aspect。哦，在Spring里面叫做Advisor，呵呵。不过，除了可以指定相应的Advisor之外，还可以使用如下代码，直接指定各种类型的Advice。
  ```
  weaver.addAdvice(...);
  ```
  - 对于Introduction之外的Advice类型，ProxyFactory内部就会为这些Advice构造相应的Advisor，只不过在为它们构造的Advisor中使用的Pointcut为Pointcut.TRUE，即这些“没穿衣服”的Advice将被应用到系统中所有可识别的Joinpoint处；
  - 而如果添加的Advice类型是Introduction类型，则会根据该Introduction的具体类型进行区分：如果是IntroductionInfo的子类实现，因为它本身包含了必要的描述信息，框架内部会为其构造一个DefaultIntroductionAdvisor；而如果是DynamicIntroductionAdvice的子类实现，框架内部将抛出AopConfigException异常（因为无法从DynamicIntroductionAdvice取得必要的目标对象信息）。

但是，在不同的应用场景下，我们可以指定更多ProxyFactory的控制属性，以便让ProxyFactory帮我们生成必要的代理对象。我们知道，Spring AOP在使用代理模式实现AOP的过程中采用了动态代理和CGLIB两种机制，分别对实现了某些接口的目标类和没有实现任何接口的目标类进行代理，所以，在使用ProxyFactory对目标类进行代理的时候，会通过ProxyFactory的某些行为控制属性对这两种情况进行区分。

在继续下面内容之前，有必要先设定一个简单的场景，以便大家结合实际情况来查看和分析在不同场景下，ProxyFactory在使用方式上的细微差异。假设我们的目标类型定义如下：

```
public interface ITask {
  void execute(TaskExecutionContext ctx);
}

public class MockTask implements ITask {

  public void execute(TaskExecutionContext ctx) {
    System.out.println("task executed.");
  }
}
```

有了要拦截的目标类，还得有织入到Joinpoint处的横切逻辑，也就是要用到某个Advice实现。我们就把之前的PerformanceMethodInterceptor先拿来一用（见代码清单9-23）。

代码清单9-23　PerformanceMethodInterceptor定义

```
public class PerformanceMethodInterceptor implements MethodInterceptor {
  private final Log logger = LogFactory.getLog(this.getClass());
  public Object invoke(MethodInvocation invocation) throws Throwable {
    StopWatch watch = new StopWatch();
    try
    {
      watch.start();
      return invocation.proceed();
    }
    finally
    {
      watch.stop();
```

```
            if(logger.isInfoEnabled())
            {
               logger.info(watch.toString);
            }
         }
      }
   }
```

有了这些之后，让我们来看一下使用ProxyFactory对实现了ITask接口的目标类，以及没有实现任何接口的目标类如何进行代理。

1. 基于接口的代理

MockTask实现了ITask接口，要对这种实现了某些接口的目标类进行代理，我们可以为ProxyFactory明确指定代理的接口类型，如下所示：

```
MockTask task = new MockTask();
ProxyFactory weaver = new ProxyFactory(task);
weaver.setInterfaces(new Class[]{ITask.class});
NameMatchMethodPointcutAdvisor advisor = new NameMatchMethodPointcutAdvisor();
advisor.setMappedName("execute");
advisor.setAdvice(new PerformanceMethodInterceptor());
weaver.addAdvisor(advisor);
ITask proxyObject =(ITask)weaver.getProxy();
proxyObject.execute(null);
```

通过setInterfaces()方法可以明确告知ProxyFactory，我们要对ITask接口类型进行代理。另外，在这里，我们通过NameMatchMethodPointcutAdvisor来指定Pointcut和相应的Advice（PerformanceMethodInterceptor）。至于什么类型的Pointcut、Advice以及Advisor，我们完全可以根据个人的喜好或者具体场景来使用，举一反三嘛！

不过，如果没有其他行为属性的干预，我们也可以不使用setInterfaces()方法明确指定具体的接口类型。这样，默认情况下，ProxyFactory只要检测到目标类实现了相应的接口，也会对目标类进行基于接口的代理，如下所示：

```
MockTask task = new MockTask();
ProxyFactory weaver = new ProxyFactory(task);
NameMatchMethodPointcutAdvisor advisor = new NameMatchMethodPointcutAdvisor();
advisor.setMappedName("execute");
advisor.setAdvice(new PerformanceMethodInterceptor());
weaver.addAdvisor(advisor);
ITask proxyObject =(ITask)weaver.getProxy();
proxyObject.execute(null);
```

这两种形式最终的结果是等效的：

```
task executed.
60 [main] INFO ...PerformanceMethodInterceptor  - 0:00:00.000
```

简单点儿说，如果目标类实现了至少一个接口，不管我们有没有通过ProxyFactory的setInterfaces()方法明确指定要对特定的接口类型进行代理，只要不将ProxyFactory的optimize和proxyTargetClass两个属性的值设置为true（这两个属性稍后将谈到），那么ProxyFactory都会按照面向接口进行代理。

看你是否真正了解了动态代理

在我们对MockTask实例进行代理之后，我们通过如下代码将取得的代理对象强制转型为

ITask，然后执行execute，那么，将取得代理对象强制转型为MockTask是否可以呢？

```
ITask proxyObject =(ITask)weaver.getProxy();
```

如果你不确定，可以把这行代码改一下试试。呵呵，答案是不可以，程序最终将抛出java.lang.ClassCastException异常。我想，你已经猜到问题出在哪里了。

请回头看一下图8-1，在代理模式的场景中，接口的具体实现类和这个具体实现类的代理对象是两个不同的对象，我们可以将接口实现类和它的代理对象都强制转型为接口类型，但是无法将代理对象类型强制转型为接口实现类类型。到我们的场景中就是，MockTask可以强制转型为ITask，MockTask的代理对象也可以强制转型为ITask，但是，要将代理对象强制转型为MockTask，一定会出问题的。

如果我们在代码最后再添加如下一行代码，来查看代码中proxyObject的类型的话：

```
System.out.println(proxyObject.getClass());
```

就会发现结果是：

```
class $Proxy0
```

(MockTask)java.lang.reflect.Proxy这样的强制转型，你觉得能行吗？

2. 基于类的代理

如果目标类没有实现任何接口，那么，默认情况下，ProxyFactory会对目标类进行基于类的代理，即使用CGLIB。假设我们现在有一个对象，定义如下：

```
public class Executable {
  public void execute(){
    System.out.println("Executable without any Interfaces");
  }
}
```

如果使用Executable作为目标对象类，那么，ProxyFactory就会对其进行基于类的代理，如下代码演示了使用ProxyFactory对Executable进行织入的过程：

```
ProxyFactory weaver = new ProxyFactory(new Executable());
NameMatchMethodPointcutAdvisor advisor = new NameMatchMethodPointcutAdvisor();
advisor.setMappedName("execute");
advisor.setAdvice(new PerformanceMethodInterceptor());
weaver.addAdvisor(advisor);
Executable proxyObject =(Executable)weaver.getProxy();
proxyObject.execute();
System.out.println(proxyObject.getClass());
```

从输出结果我们也可以看出来，最终的代理对象是基于CGLIB的：

```
Executable without any Interfaces
2143 [main] INFO ...PerformanceMethodInterceptor  - 0:00:00.000
class ...Executable$$EnhancerByCGLIB$$9e62fc83
```

但是，即使目标对象类实现了至少一个接口，我们也可以通过proxyTargetClass属性强制ProxyFactory采用基于类的代理。以MockTask为例，它实现了ITask接口，默认情况下ProxyFactory对其会采用基于接口的代理，但是，通过proxyTargetClass，我们可以改变这种默认行为（见如下代码）：

```
ProxyFactory weaver = new ProxyFactory(new MockTask());
```

```
weaver.setProxyTargetClass(true);
NameMatchMethodPointcutAdvisor advisor = new NameMatchMethodPointcutAdvisor();
advisor.setMappedName("execute");
advisor.setAdvice(new PerformanceMethodInterceptor());
weaver.addAdvisor(advisor);
MockTask proxyObject =(MockTask)weaver.getProxy();
proxyObject.execute(null);
System.out.println(proxyObject.getClass());
```

现在，我们可以直接将代理对象强制转型为MockTask类型，并且，从输出结果也可以看到，最终的代理对象是基于CGLIB的，而不是动态代理的：

```
task executed.
311 [main] INFO ...PerformanceMethodInterceptor  - 0:00:00.000
class ...MockTask$$EnhancerByCGLIB$$4bf6056
```

除此之外，如果将ProxyFactory的optimize属性设定为true的话，ProxyFactory也会采用基于类的代理机制。关于optimize属性的更多信息，我们将在后面给出。

总地来说，如果满足以下列出的三种情况中的任何一种，ProxyFactory将对目标类进行基于类的代理。

- 如果目标类没有实现任何接口，不管proxyTargetClass的值是什么，ProxyFactory会采用基于类的代理。
- 如果ProxyFactory的proxyTargetClass属性值被设置为true，ProxyFactory会采用基于类的代理。
- 如果ProxyFactory的optimize属性值被设置为true，ProxyFactory会采用基于类的代理。

3. Introduction的织入

之所以将Introduction的织入单独列出，是因为Introduction型Advice比较特殊，如下所述。

- Introduction可以为已经存在的对象类型添加新的行为，只能应用于对象级别的拦截，而不是通常Advice的方法级别的拦截，所以，进行Introduction的织入过程中，不需要指定Pointcut，而只需要指定目标接口类型。
- Spring的Introduction支持只能通过接口定义为当前对象添加新的行为，所以，我们需要在织入的时机，指定新织入的接口类型。

鉴于以上两点，使用ProxyFactory进行Introduction的织入代码示例如代码清单9-24所示。

代码清单9-24 使用ProxyFactory进行Introduction的织入过程示例

```
ProxyFactory weaver = new ProxyFactory(new Developer());
weaver.setInterfaces(new Class[]{IDeveloper.class,ITester.class});
TesterFeatureIntroductionInterceptor advice = new TesterFeatureIntroductionInterceptor();
weaver.addAdvice(advice);
//DefaultIntroductionAdvisor advisor = new DefaultIntroductionAdvisor(advice,advice);
//weaver.addAdvisor(advisor);

Object proxy = weaver.getProxy();
((ITester)proxy).testSoftware();
((IDeveloper)proxy).developSoftware();
```

如果我们不使用Advisor而直接为ProxyFactory指定Advice的话，还记得ProxyFactory会如何处理的嘛？ProxyFactory会在自身内部构建相应的Advisor来使用，对吧？因为TesterFeature-IntroductionInterceptor是IntroductionInfo的子类，所以，ProxyFactory内部会创建一个默认的DefaultIntroductionAdvisor实例，就跟我们注释掉的两行代码效果一样。

对Introduction进行织入，与基于接口的代理形式有点像，但有少许差异。对Introduction进行织入，新添加的接口类型必须是通过setInterfaces指定的，而原来的目标对象，是采用基于接口的代理形式还是采用基于类的代理形式，完全是可以自由选择的。上面我们通过setInterfaces同时指定了目标对象实现的接口和新添加的接口类型，在进行Introduction织入的同时使用了基于接口的代理形式。我们同样可以在织入Introduction的同时，使用基于类的代理形式（见代码清单9-25）。

代码清单9-25 使用ProxyFactory进行基于类的代理方式的Introduction织入过程示例

```
ProxyFactory weaver = new ProxyFactory(new Developer());
weaver.setProxyTargetClass(true);
weaver.setInterfaces(new Class[]{ITester.class});
TesterFeatureIntroductionInterceptor advice = new TesterFeatureIntroductionInterceptor();
weaver.addAdvice(advice);
//DefaultIntroductionAdvisor advisor = new DefaultIntroductionAdvisor(advice,advice);
//weaver.addAdvisor(advisor);

Object proxy = weaver.getProxy();
((ITester)proxy).testSoftware();
((Developer)proxy).developSoftware();
```

我们通过weaver.setProxyTargetClass(true);强制使用了基于类的代理，所以，现在得将代理对象转型为Developer而不是IDeveloper。

从介绍Advice类型到介绍Advisor类型，针对Introduction的部分都是单独陈述的，或许你已经猜到，Introduction的Advice以及Advisor是不能跟其他Advice和Advisor混用的，要织入Introduction，你只能使用IntroductionAdvisor或者其子类，而不能使用其他的组合。

9

9.5.2 看清 ProxyFactory 的本质

知其表而不知其里，充其量你只能算一个画匠，而不是画师；只懂得如何使用API，而不知道这些API为何如此设计，使你迈不出从“画匠”到“画师”的那一步，如果你想迈出这一步，那不妨随我看一下这ProxyFactory内部到底有何“猫腻儿”，何如？

认识ProxyFactory的本质，不仅可以让我们清楚它如何实现，帮助我们在以后的系统设计中吸取宝贵的经验，而且可以进一步帮助我们更好地使用ProxyFactory。

> **注意** 因为本节剩下的内容涉及的都是ProxyFactory或者Spring AOP框架的实现，所以，大部分类全部来自org.springframework.aop.framework包。

要了解ProxyFactory，我们得先从它的“根”说起，即org.springframework.aop.framework.AopProxy，该接口定义如下：

```
public interface AopProxy {
  Object getProxy();
  Object getProxy(ClassLoader classLoader);
}
```

Spring AOP框架内使用AopProxy对使用的不同的代理实现机制进行了适度的抽象，针对不同的代理实现机制提供相应的AopProxy子类实现。目前，Spring AOP框架内提供了针对JDK的动态代理和CGLIB两种机制的AopProxy实现（见图9-10）。

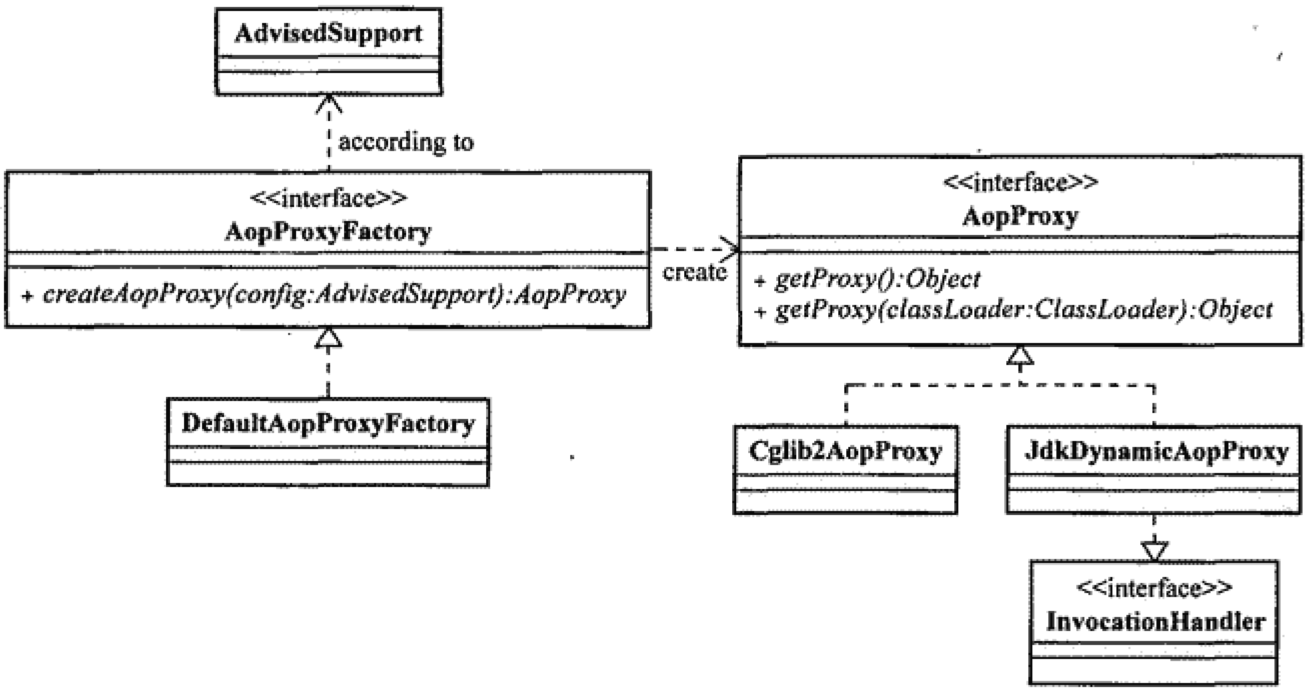

图9-10　AopProxy相关结构图

当前，AopProxy有Cglib2AopProxy和JdkDynamicAopProxy两种实现。因为动态代理需要通过InvocationHandler提供调用拦截，所以，JdkDynamicAopProxy同时实现了InvocationHandler接口。不同AopProxy实现的实例化过程采用工厂模式（确切地说是抽象工厂模式）进行封装，即通过org.springframework.aop.framework.AopProxyFactory进行。AopProxyFactory接口的定义如下所示：

```
public interface AopProxyFactory {
   AopProxy createAopProxy(AdvisedSupport config) throws AopConfigException;
}
```

AopProxyFactory根据传入的AdvisedSupport实例提供的相关信息，来决定生成什么类型的AopProxy。不过，具体工作会转交给AopProxyFactory的具体实现类。而实际上这个AopProxyFactory实现类现在就一个，即org.springframework.aop.framework.DefaultAopProxyFactory。

DefaultAopProxyFactory的实现逻辑很简单，如以下伪代码所示：

```
if (config.isOptimize() || config.isProxyTargetClass() || hasNoUserSuppliedProxyInterfaces (config))
{
   // 创建Cglib2AopProxy实例，并返回;
}
else
{
   // 创建JdkDynamicAopProxy实例，并返回;
}
```

也就是说，如果传入的AdvisedSupport实例config的isOptimize或者isProxyTargetClass方法返回true，或者目标对象没有实现任何接口，则采用CGLIB生成代理对象，否则使用动态代理。还记得ProxyFactory会采用基于类的代理形式生成代理对象需要满足的条件吗？这里是一个关键点，但是走到这里，你还是无法理解为什么ProxyFactory会有这种好像偶然的行为。别急，我们接着看！

AopProxyFactory需要根据createAopProxy方法传入的AdvisedSupport实例信息，来构建相应的AopProxy。下面我们要看看这个AdvisedSupport到底是何方神圣。

说得简单一点儿，AdvisedSupport其实就是一个生成代理对象所需要的信息的载体，该类相关的类层次图，见图9-11。

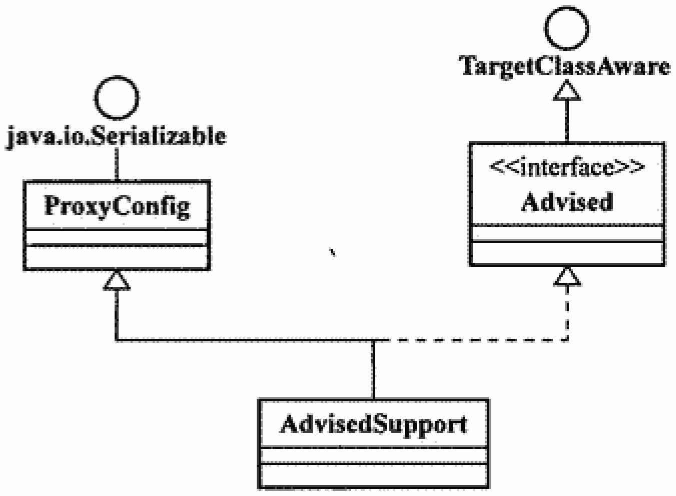

图9-11 AdvisedSupport类层次图

AdvisedSupport所承载的信息可以划分为两类，一类以org.springframework.aop.framework.ProxyConfig为统领，记载生成代理对象的控制信息；一类以org.springframework.aop.framework.Advised为旗帜，承载生成代理对象所需要的必要信息，如相关目标类、Advice、Advisor等。

ProxyConfig其实就是一个普通的JavaBean，它定义了5个boolean型的属性，分别控制在生成代理对象的时候，应该采取哪些行为措施，下面是这5个属性的详细情况。

- proxyTargetClass。在9.5.1节的第2小节中已经提到过这个属性，如果proxyTargetClass属性设置为true，则ProxyFactory将会使用CGLIB对目标对象进行代理，默认值为false。
- optimize。该属性的主要用于告知代理对象是否需要采取进一步的优化措施，如代理对象生成之后，即使为其添加或者移除了相应的Advice，代理对象也可以忽略这种变动。另外，我们也曾经提到，当该属性为true时，ProxyFactory会使用CGLIB进行代理对象的生成。默认情况下，该属性为false。更多信息参照Spring的Javadoc以及参考文档。
- opaque。该属性用于控制生成的代理对象是否可以强制转型为Advised，默认值为false，表示任何生成的代理对象都可强制转型为Advised，我们可以通过Advised查询代理对象的一些状态。
- exposeProxy。设置exposeProxy，可以让Spring AOP框架在生成代理对象时，将当前代理对象绑定到ThreadLocal。如果目标对象需要访问当前代理对象，可以通过AopContext.currentProxy()取得。该属性的用途将在后文中详细讲述。出于性能方面考虑，该属性默认值为false。
- frozen。如果将frozen设置为true，那么一旦针对代理对象生成的各项信息配置完成，则不容许更改。比如，如果ProxyFactory的设置完毕，并且fronzen为true，则不能对Advice进行任何变动，这样可以优化代理对象生成的性能。默认情况下，该值为false。

要生成代理对象，只有ProxyConfig提供的控制信息还不够，我们还需要生成代理对象的一些具体信息，比如，要针对哪些目标类生成代理对象，要为代理对象加入何种横切逻辑等，这些信息可以通过org.springframework.aop.framework.Advised设置或者查询。默认情况下，Spring AOP框架返回的代理对象都可以强制转型为Advised，以查询代理对象的相关信息。Advised的接口定义代码

太长，我们就不在此罗列了，你可以参照它的Javadoc。简单点儿说，我们可以使用Advised接口访问相应代理对象所持有的Advisor，进行添加Advisor、移除Advisor等相关动作。即使代理对象已经生成完毕，也可对其进行这些操作。直接操作Advised，更多时候用于测试场景，可以帮助我们检查生成的代理对象是否如所期望的那样。（有关Advised的更多信息，请参照Spring的参考文档，因为与我们的主题相关性不大，这里不进行详细讲述。）

回到之前的AdvisedSupport话题，AdvisedSupport继承了ProxyConfig，我们可以通过AdvisedSupport设置代理对象生成的一些控制属性。AdvisedSupport同时实现了Advised接口，我们也可以通过AdvisedSupport设置生成代理对象相关的目标类、Advice等必要信息。这样，具体的AopProxy实现在生成代理对象时，可以从AdvisedSupport这里取得所有这些必要信息。

现在回到主题ProxyFactory。AopProxy、AdvisedSupport与ProxyFactory是什么关系呢？先看图9-12。

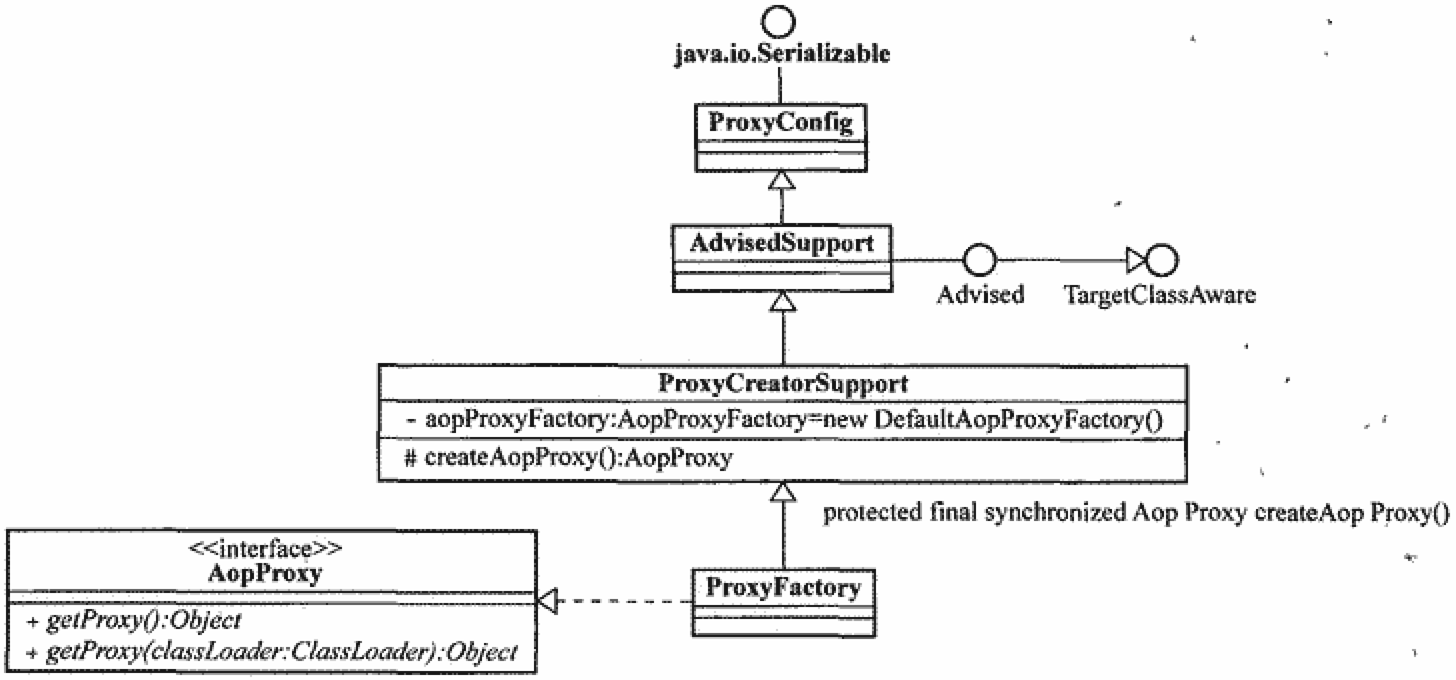

图9-12　ProxyFactory继承层次类图

ProxyFactory集AopProxy和AdvisedSupport于一身，所以，可以通过ProxyFactory设置生成代理对象所需要的相关信息，也可以通过ProxyFactory取得最终生成的代理对象。前者是AdvisedSupport的职责，后者是AopProxy的职责。

为了重用相关逻辑，Spring AOP框架在实现的时候，将一些公用的逻辑抽取到了org.springframework.aop.framework.ProxyCreatorSupport中，它自身就继承了AdvisedSupport，所以，生成代理对象的必要信息从其自身就可以搞到。为了简化子类生成不同类型AopProxy的工作，ProxyCreatorSupport内部持有一个AopProxyFactory实例，默认采用的是DefaultAopProxyFactory（也可以通过构造方法或者setter方法设置其他实现，如果有的话）。DefaultAopProxyFactory的默认行为前面已经讲述过了。ProxyFactory作为一个ProxyCreatorSupport自然继承了这种行为，从它的使用中我们已经领略过了。

前面已经说过了，ProxyFactory只是Spring AOP中最基本的织入器实现。实际上，ProxyFactory还有几个“兄弟”，这从ProxyCreatorSupport的继承类图（图9-13）中可以看到。

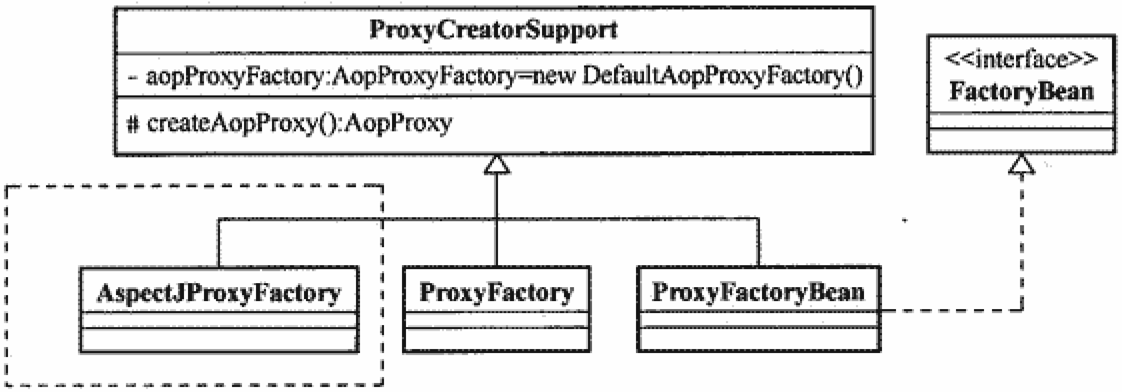

图9-13 ProxyFactory的“兄弟”

后文中将详细讲述AspectJProxyFactory。当前，我们还是先来看看ProxyFactoryBean。

9.5.3 容器中的织入器——ProxyFactoryBean

虽然使用ProxyFactory，可以让我们能够独立于Spring的IoC容器之外来使用Spring的AOP支持，但是，将Spring AOP与Spring的IoC容器支持相结合，才是发挥Spring AOP更大作用的最佳途径。通过结合Spring的IoC容器，我们可以在容器中对Pointcut和Advice等进行管理，即使它们依赖于其他业务对象，也可以很容易地注入其中。

在IoC容器中，使用org.springframework.aop.framework.ProxyFactoryBean作为织入器，它的使用与ProxyFactory无太大差别。不过在演示ProxyFactoryBean的使用之前，我们有必要在看清了ProxyFactory本质的前提下，进一步弄明白ProxyFactoryBean的本质。

1. ProxyFactoryBean的本质

对于ProxyFactoryBean，我们应该这样断词，即Proxy+FactoryBean，而不是ProxyFactory+ Bean。也就是说，ProxyFactoryBean本质上是一个用来生产Proxy的FactoryBean。还记得IoC容器中的FactoryBean的作用吧？如果容器中的某个对象持有某个FactoryBean的引用，它取得的不是FactoryBean本身，而是FactoryBean的getObject()方法所返回的对象。所以，如果容器中某个对象依赖于ProxyFactoryBean，那么它将会使用到ProxyFactoryBean的getObject()方法所返回的代理对象，这就是ProxyFactryBean得以在容器中游刃有余的原因。

要让ProxyFactoryBean的getObject()方法返回相应目标对象类的代理对象其实很简单。因为ProxyFactoryBean继承了与ProxyFactory共有的父类ProxyCreatorSupport，而ProxyCreator-Support基本上已经把要做的事情（如设置目标对象、配置其他部件、生成对应的AopProxy等）全部完成了。我们只需在ProxyFactoryBean的getObject()方法中通过父类的createAopProxy()取得相应的AopProxy，然后“return AopProxy.getProxy()”即可。

因为涉及FactoryBean，所以在实现getObject()时，逻辑上还得点缀一下。我们来看ProxyFactoryBean的getObject()定义（见代码清单9-26）。

代码清单9-26 ProxyFactoryBean的getObject()方法逻辑

```
public Object getObject() throws BeansException {
  initializeAdvisorChain();
  if (isSingleton()) {
    return getSingletonInstance();
  }
  else
  {
```

```
        if (this.targetName == null)
        {
          logger.warn("Using non-singleton proxies with singleton targets is often undesirable." +➥
          "Enable prototype proxies by setting the 'targetName' property.");
        }
        return newPrototypeInstance();
        }
    }
```

FactoryBean定义中要求标明返回的对象是以singleton的scope返回，还是以prototype的scope返回。所以，得针对这两种情况分别返回不同的代理对象，以满足FactoryBean的isSingleton()方法的语义。

如果将ProxyFactoryBean的singleton属性设置为true，则ProxyFactoryBean在第一次生成代理对象之后，会通过内部实例变量singletonInstance（Object类型）缓存生成的代理对象。之后，所有的请求将会返回这一缓存实例，从而满足singleton的语义。反之，如果将ProxyFactoryBean的singleton属性设置为false，那么，ProxyFactoryBean每次都会重新检测各项设置，并为当前调用准备一套新的环境，然后再根据最新的环境数据，返回一个新的代理对象。因此，如果singleton属性为false，在生成代理对象的性能上存在损失。如果非要这么做，请确保有充足的理由。singleton默认值为true，即返回同一个代理对象实例。

如果对ProxyFactoryBean的细节感兴趣，可以读一下ProxyFactoryBean的代码。

2. ProxyFactoryBean的使用

与ProxyFactory一样，通过ProxyFactoryBean，我们可以在生成目标对象的代理对象的时候，指定使用基于接口的代理还是基于类的代理方式，而且，因为它们全部继承自同一个父类，大部分可设置项目都相同。不过，ProxyFactoryBean在继承了父类ProxyCreatorSupport的所有配置属性之外，还添加了几个自己独有的，如下所示。

- proxyInterfaces。如果我们要采用基于接口的代理方式，那么需要通过该属性配置相应的接口类型，这是一个Collection类型实例，所以我们可以通过配置元素<list>来指定一个或者多个接口类型。实际上，这与通过Interfaces属性指定接口类型是等效的，我们完全可以随个人喜好来使用，虽然使用proxyInterfaces可以保持使用上的统一风格。另外，如果目标对象实现了某个或者多个接口，即使我们不通过该属性指定要代理的接口类型，ProxyFactroyBean也可以自动检测到目标对象所实现的接口，并对其进行基于接口的代理。因为ProxyFactoryBean有一个autodetectInterfaces属性，该属性默认值为true，即如果没有明确指定要代理的接口类型，ProxyFactoryBean会自动检测目标对象所实现的接口类型并进行代理。
- interceptorNames。通过该属性，我们可以指定多个将要织入到目标对象的Advice、拦截器以及Advisor，而再也不用通过ProxyFactory那样的addAdvice或者addAdvisor方法一个一个地添加了。因为该属性属于Collection类型，所以通常我们会使用配置元素<list>添加需要的拦截器名称。该属性有两个特性需要提及，如以下所述。
  - 如果没有通过相应的设置目标对象的方法明确为ProxyFactoryBean设置目标对象，那么可以在interceptorNames的最后一个元素位置，放置目标对象的bean定义名称。这是个特例，大部分情况下，还是建议明确指定目标对象，而避免这种配置方式。
  - 通过在指定的interceptorNames某个元素名称之后添加*通配符，可以让ProxyFactory-Bean在容器中搜寻符合条件的所有的Advisor并应用到目标对象。这些符合条件的Advisor，Spring参考文档中称之为global advisor。代码清单9-27给出了这种用法的示例。

- singleton。因为ProxyFactoryBean本质上是一个FactoryBean，所以我们可以通过singleton属性，指定每次getObject调用是返回同一个代理对象，还是返回一个新的。通常情况下是返回同一个代理对象，即singleton为true。只有在需要返回有状态的代理对象的情况下，才会将singleton设置为false，如使用Introduction的场合。

代码清单9-27 包含*通配符的interceptorNames属性使用示例

```
<bean id="proxy"
  class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="target" ref="..."/>
  <property name="interceptorNames">
    <list>
      <value>global*</value>
    </list>
  </property>
</bean>
<bean id="global_debug" ➥
class="org.springframework.aop.interceptor.DebugInterceptor"/>
<bean id="global_performance" class="org.springframework.aop.interceptor. ➥
PerformanceMonitorInterceptor"/>
```

要在容器中通过ProxyFactoryBean使用基于接口的代理方式，通常可以采用代码清单9-28所示的配置方式。

代码清单9-28 通过ProxyFactoryBean使用基于接口的代理方式的配置示例

```
<bean id="pointcut" class="org.springframework.aop.support.NameMatchMethodPointcut">
  <property name="mappedName" value="execute"/>
</bean>

<bean id="performanceInterceptor" class="...advice.PerformanceMethodInterceptor">
</bean>

<bean id="performanceAdvisor" class="org.springframework.aop.support.DefaultPointcutAdvisor">
  <property name="pointcut">
    <ref bean="pointcut"/>
  </property>
  <property name="advice">
    <ref bean="performanceInterceptor"/>
  </property>
</bean>

<bean id="task" class="...MockTask">
</bean>

<bean id="taskProxy" class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="target">
    <ref bean="task"/>
  </property>
  <property name="proxyInterfaces">
    <list>
      <value>...ITask</value>
    </list>
  </property>
  <property name="interceptorNames">
    <list>
      <value>performanceAdvisor</value>
```

9

```
      </list>
    </property>
  </bean>
```

现在，从Pointcut到Advice再到Advisor，从目标对象到相应的代理对象，全部都由IoC容器统一管理。为ProxyFactoryBean指定目标对象、要代理的接口类型以及相应的Advisor或Advice，ProxyFactoryBean就会返回目标对象的代理对象供调用者使用。我们可以将生成的代理对象直接注入到依赖的主体对象中，但是这里有一个初学者容易犯的错误，就是通常会将目标对象task注入依赖的主体对象，而不是目标对象的代理对象taskProxy。通过之前有关代理模式的讲解，现在应该不会犯这种错误了。将没有织入任何横切逻辑的目标对象，而不是代理对象注入依赖的主体对象，一定不会产生任何拦截效果。为了避免这种问题，如果没有依赖于目标对象的依赖关系，可以将目标对象的bean定义声明为内部bean，这样，就不会出现该引用目标对象代理对象的地方，反而因不慎或者其他原因而引用目标对象本身的情况。代码清单9-29演示了这种好的实践方式。

代码清单9-29　使用内部bean定义避免错误的依赖注入引用

```
...

<bean id="taskProxy" class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="target">
    <bean class="...MockTask"/>
  </property>
  <property name="proxyInterfaces">
    <list>
      <value>...ITask</value>
    </list>
  </property>
  <property name="interceptorNames">
    <list>
      <value>performanceAdvisor</value>
    </list>
  </property>
</bean>
```

因为autodetectInterfaces的默认值为true，如果确认目标对象所实现的接口就是要代理的接口，那么，完全可以省略通过interfaces或者proxyInterfaces明确指定代理接口的配置。代码清单9-29的配置内容可以精简如下：

```
...

<bean id="taskProxy" class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="target">
    <bean class="...MockTask"/>
  </property>
  <property name="interceptorNames">
    <list>
      <value>performanceAdvisor</value>
    </list>
  </property>
</bean>
```

如果没有指定要代理的接口类型，并且目标对象也没有实现任何接口，那么，ProxyFactoryBean会采用基于类的代理方式为目标对象生成代理对象。不过，即使目标对象实现了某些接口，我们也可以强制ProxyFactoryBean采用基于类的代理方式来生成代理对象。与ProxyFactory一样，只要指定proxyTargetClass为true就可以了（见代码清单9-30）。

代码清单9-30 强制ProxyFactoryBean使用基于类的代理方式的配置示例

```
...
<bean id="taskProxy" class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="target">
    <bean class="...MockTask"/>
  </property>
  <property name="proxyTargetClass">
    <value>true</value>
  </property>
  <property name="interceptorNames">
    <list>
      <value>performanceAdvisor</value>
    </list>
  </property>
</bean>
```

不过，现在客户端代码不能将代理对象强制转型为ITask，而应该强制转型为目标对象的具体类型，即MockTask，如下所示：

```
ApplicationContext ctx = ...;
// ITask task = (ITask)ctx.getBean("taskProxy");
// 错误!

MockTask task = (MockTask)ctx.getBean("taskProxy");
task.execute(null);
...
```

> **提示** 有时，我们的应用可能需要依赖于第三方库，这些库中可能有些对象是出于简单实用的目的，就是没有进行面向接口编程，自然就没有实现任何接口。而且，我们自己设计和实现的类，可能出于某种目的，也是没有实现接口的必要，这时，就需要通过将proxyTargetClass设置为true来解决代理的问题。

9

说完了如何通过ProxyFactoryBean生成目标对象的代理对象（使用“基于接口的代理”方式也好，使用“基于类的代理”方式也好）。下面该说一下Introduction的代理了，因为它一直比较特立独行嘛！

为了演示Introduction的织入，我们引入一个ICounter接口定义以及一个简单实现类，然后将这个接口的行为和状态添加到ITask相应实现类中。ICounter接口以及相关实现类定义见代码清单9-31。

代码清单9-31 ICounter接口以及相关实现类定义

```
public interface ICounter {
  void resetCounter();
  int getCounter();
}

public class CounterImpl implements ICounter {
  private int counter;

  public int getCounter() {
    counter++;
    return counter;
  }

  public void resetCounter() {
```

```
        counter = 0;
    }
}
```

要将ICounter的行为添加到ITask相应实现类中，可以采用代码清单9-32所示的配置。

代码清单9-32　将ICounter行为添加到ITask的配置示例

```
<bean id="task" class="...MockTask" singleton="false">
</bean>

<bean id="introducedTask" class="org.springframework.aop.framework.ProxyFactoryBean" ➥
singleton="false">
   <property name="targetName">
      <value>task</value>
   </property>
   <property name="proxyInterfaces">
      <list>
         <value>...ITask</value>
         <value>...ICounter</value>
      </list>
   </property>
   <property name="interceptorNames">
      <list>
         <value>introductionInterceptor</value>
      </list>
   </property>
</bean>

<bean id="introductionInterceptor" ➥
class="org.springframework.aop.support.DelegatingIntroductionInterceptor"
singleton="false">
   <constructor-arg>
      <bean class="...CounterImpl">
      </bean>
   </constructor-arg>
</bean>
```

请注意，我们将目标对象的bean定义、ProxyFactoryBean的bean定义，以及相应Introduction-Interceptor的bean定义的scope，全部声明为prototype，也就是singleton="false"，并且，这种情况下，我们使用的是"taskName"而不是"task"来指定目标对象（使用task通过ref指定prototype类型的依赖会有什么效果，在Spring的IoC容器部分已经讲述过了。）。这样才能保证每次取得的代理对象都持有各自独有的状态和行为，如下是调用执行的代码示例：

```
ApplicationContext ctx = new ClassPathXmlApplicationContext("...");
Object proxy1 = ctx.getBean("introducedTask");
Object proxy2 = ctx.getBean("introducedTask");

System.out.println(((ICounter)proxy1).getCounter());
System.out.println(((ICounter)proxy1).getCounter());
System.out.println(((ICounter)proxy2).getCounter());
```

因为proxy1和proxy2各自拥有独立的状态，所以，输出为:

```
1
2
1
```

我们之前说过，DelegatingIntroductionInterceptor是一个“伪军”，如果不是采用prototype

的scope为每一个代理对象都分配一个该类型实例，则无法保证各代理对象拥有各自的状态。不过，如果使用DelegatePerTargetObjectIntroductionInterceptor，那么可以共用一个该类型的Advice实例（即使用sigleton的scope），见代码清单9-33。

代码清单9-33 使用DelegatePerTargetObjectIntroductionInterceptor代替DelegatingIntroductionInterceptor后的配置实例

```
<bean id="task" class="...MockTask" singleton="false">
</bean>

<bean id="introducedTask" class="org.springframework.aop.framework.ProxyFactoryBean" ➥
singleton="false">
  <property name="target"><ref bean="task"/></property>
  <property name="proxyInterfaces">
    <list>
      <value>...ITask</value>
      <value>...ICounter</value>
    </list>
  </property>
  <property name="interceptorNames">
    <list>
      <value>introductionInterceptor</value>
    </list>
  </property>
</bean>

<bean id="introductionInterceptor" class="org.springframework.aop.support. ➥
DelegatePerTargetObjectIntroductionInterceptor">
  <constructor-arg index="0">
    <value>...CounterImpl</value>
  </constructor-arg>
  <constructor-arg index="1">
    <value>...ICounter</value>
  </constructor-arg>
</bean>
```

至于你的自定义IntroductionInterceptor，在应用的时候，请根据情况设置IntroductionInterceptor的scope以保证状态的独立性。有关ProxyFactoryBean的更多配置项细节，请参照对应的Javadoc，这里就不赘述了。我们得加快织入的速度了，毕竟，一个一个地配置ProxyFactoryBean可不是什么令人感到轻松、愉快的事情。

9.5.4 加快织入的自动化进程

在IoC容器中使用ProxyFactoryBean进行横切逻辑的织入固然不错，但是，我们都是针对每个目标对象，然后给出它们各自所对应的ProxyFactoryBean配置。如果目标对象就那么几个，那还应付得过来。但系统中那么多的业务对象可能都是目标对象，如果还是用ProxyFactoryBean一个个地进行配置，估计得累得吐血，所以，我们得寻求更加简单快捷的方式。

> **注意** 当然，如果系统小，而且横切关注点不多，目标对象少，那么简单地使用ProxyFactoryBean也不失为合适的方式。

Spring AOP给出了自动代理（AutoProxy）机制，用以帮助我们解决使用ProxyFactoryBean配置工作量比较大的问题。

1. 自动代理得以实现的原理

要使用自动代理机制，需要以Spring的IoC容器为依托。更进一步说，需要使用ApplicationContext类型的IoC容器。虽然可以通过进一步的编码，让BeanFactory也支持这一功能，但是既然要自动，还是一步到位的好。

Spring AOP的自动代理的实现建立在IoC容器的BeanPostProcessor概念之上。还记得我们可以使用BeanPostProcessor干预bean的实例化过程吗？通过BeanPostProcessor，我们可以在遍历容器中所有bean的基础上，对遍历到的bean进行一些操作。有了这个前提条件，要实现自动代理就很容易了。

其实我们不难想到，只要提供一个BeanPostProcessor，然后在这个BeanPostProcessor内部实现这样的逻辑，即当对象实例化的时候，为其生成代理对象并返回，而不是实例化后的目标对象本身，从而达到代理对象自动生成的目的。该逻辑如果以伪代码演示，如代码清单9-34所示。

代码清单9-34　自动代理实现原理的伪代码示例

```
for(bean in IoC container)
{
   检查当前bean定义是否符合拦截条件;
   如果符合拦截条件，则
   {
      Object proxy = createProxyFor(bean);
      return proxy;
   }
   否则
   {
      Object instance = createInstance(bean);
      return instance;
   }
}
```

而对于Object proxy = createProxyFor(bean);这行代码，如何根据目标对象生成相应的代理对象的细节，我想就不用说了吧（不就是使用ProxyFactory或者ProxyFactoryBean嘛）！我要说的是第一行，即检查当前bean定义是否符合拦截条件。

要检查当前bean是否符合拦截条件，首先需要知道拦截条件是什么，那么我们就要通过某种方式，告知具体的自动代理实现类都有哪些拦截条件：

- 可以通过外部配置文件传入这些拦截条件信息，比如我们在容器的配置文件中注册的有关Pointcut以及Advisor等就包括这些信息；
- 还可以在具体类的定义文件中，通过元数据来指明具体的拦截条件是什么，比如可以通过Jakarta Commons Attributes或者Java 5的注解，直接在代码类中标注Pointcut等拦截信息。

但不管采用什么方式来提供拦截信息，我们都可以提供相应的自动代理实现类来读取这些信息，并为相应的目标对象自动生成代理对象。

下面，让我们先看一下Spring AOP都提供了哪些可以使用的自动代理实现类，免得做一些无用功。

2. 可用的AutoProxyCreator

Spring AOP在org.springframework.aop.framework.autoproxy包中提供了两个常用的AutoProxyCreator，即BeanNameAutoProxyCreator和DefaultAdvisorAutoProxyCreator。

- BeanNameAutoProxyCreator

使用BeanNameAutoProxyCreator，我们可以通过指定一组容器内的目标对象对应的beanName，将指定的一组拦截器应用到这些目标对象之上。代码清单9-35给出了通常情况下使用BeanNameAutoProxyCreator的配置演示。

代码清单9-35 使用BeanNameAutoProxyCreator进行自动代理的配置示例

```
<bean id="target1" class="..."/>
<bean id="target2" class="..."/>

<bean id="mockTask" class="..."/>
<bean id="fakeTask" class="..."/>

<bean id="taskThrowsAdvice" class="...TaskThrowsAdvice">
</bean>

<bean id="performanceInterceptor" class="...PerformanceMethodInterceptor">
</bean>

<bean class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
  <property name="beanNames">
    <list>
      <value>target1</value>
      <value>target2</value>
    </list>
  </property>
  <property name="interceptorNames">
    <list>
      <value>taskThrowsAdvice</value>
    </list>
  </property>
</bean>

<bean class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
  <property name="beanNames">
    <list>
      <value>mockTask</value>
      <value>fakeTask</value>
    </list>
  </property>
  <property name="interceptorNames">
    <list>
      <value>performanceInterceptor</value>
    </list>
  </property>
</bean>
```

通过beanNames，我们可以指定要对容器中的哪些bean自动生成代理对象。通过interceptorNames，我们可以指定将要应用到目标对象的拦截器、Advice或者Advisor等。实际上，我们可以将以上两个BeanNameAutoProxyCreator并作一个使用。给出至少两个BeanNameAutoProxyCreator定义，只是为了表明，我们可以使用多个BeanNameAutoProxyCreator以细化横切逻辑的织入范围。

如果一类目标对象的bean定义名称相似，如可能我们系统中服务层的bean定义名称都以service结尾，那么我们可以如代码清单9-36所示，在beanNames属性中指定*通配符以简化配置。

代码清单9-36 在beanNames属性中指定*通配符以简化配置

```
...

<bean class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
  <property name="beanNames">
    <list>
      <value>target*</value>
      <value>*service</value>
```

9

```
      </list>
    </property>
    <property name="interceptorNames">
      <list>
        <value>performanceInterceptor</value>
      </list>
    </property>
</bean>
...
```

如果在使用*通配符的情况下，我们还是要指定一长串的beanNames，那么使用一下配置诀窍吧。对于String[]型的数组，我们也可以不用<list>元素，而是如下所示，直接使用逗号分隔数组的各个元素即可：

```
...

<bean class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
    <property name="beanNames">
      <value>target*,*Task,*service</value>
    </property>
    <property name="interceptorNames">
      <list>
        <value>performanceInterceptor</value>
      </list>
    </property>
</bean>
```

当然，这是容器配置中的小诀窍，与BeanNameAutoProxyCreator可没太大的关系。因为我们可以在任何类似场合使用这种配置方式。

- DefaultAdvisorAutoProxyCreator

如果把BeanNameAutoProxyCreator比作半自动步枪的话，那么DefaultAdvisorAutoProxyCreator可算全自动步枪啦。使用DefaultAdvisorAutoProxyCreator，我们只需要在ApplicationContext的配置文件中注册一下DefaultAdvisorAutoProxyCreator的bean定义就可以了。剩下的事情，DefaultAdvisorAutoProxyCreator全部搞定。代码清单9-37给出了使用DefaultAdvisorAutoProxyCreator的配置演示。

代码清单9-37　DefaultAdvisorAutoProxyCreator的使用配置示例

```
<bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"/>

<bean id="target1" class="..."/>
<bean id="target2" class="..."/>

<bean id="mockTask" class="..."/>
<bean id="fakeTask" class="..."/>

<bean id="logAdvisor" class="org.springframework.aop.support.DefaultPointcutAdvisor">
    <property name="pointcut">
      ...
    </property>
    <property name="advice">
      <bean id="performanceInterceptor" class="...PerformanceMethodInterceptor">
      </bean>
    </property>
</bean>

<bean id="logAdvisor" class="org.springframework.aop.support.DefaultPointcutAdvisor">
```

```
    <property name="pointcut">
      ...
    </property>
    <property name="advice">
      <bean id="taskThrowsAdvice" class="...TaskThrowsAdvice">
      </bean>
    </property>
  </bean>
```

将DefaultAdvisorAutoProxyCreator注册到容器后，它就会自动搜寻容器内的所有Advisor，然后根据各个Advisor所提供的拦截信息，为符合条件的容器中的目标对象生成相应的代理对象。注意，DefaultAdvisorAutoProxyCreator只对Advisor有效，因为只有Advisor才既有Pointcut信息以捕捉符合条件的目标对象，又有相应的Advice。

使用DefaultAdvisorAutoProxyCreator对容器内所有bean定义对应的对象进行自动代理之后，我们从容器中取得的对象实例，就都是代理后已经包含了织入的横切逻辑的代理对象了，除非该对象不符合Pointcut规定的拦截条件。

因为DefaultAdvisorAutoProxyCreator的处理范围比较大，所以，为了避免将不必要的横切逻辑织入到不需要的目标对象之上，应该尽量细化各个Advisor的定义，或者，转而使用BeanNameAutoProxyCreator来进行更加细粒度的织入范围控制。

提示 某些时候，如果系统中有许多目标对象类无法采用"基于接口的代理"形式进行拦截，那么我们就需要明确告知AOP框架，在生成代理对象的时候应该采用"基于类的代理"形式。使用自动代理之前，我们可能需要依次为每一个ProxyFactoryBean都指定一下proxyTargetClass属性，但是使用了自动代理之后，因为所有AutoProxyCreator的父类都继承自ProxyConfig，所以只需要通过具体的AutoProxyCreator指定一下proxyTargetClass属性，然后就可以控制所有代理对象的生成是采用"基于类的代理"形式了。对于DefaultAdvisorAutoProxyCreator来说，是如下这个样子：

```
<bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator">
  <property name="proxyTargetClass">
    <value>true</value>
  </property>
</bean>
```

9

3. 扩展AutoProxyCreator

如果BeanNameAutoProxyCreator和DefaultAdvisorAutoProxyCreator不能满足我们的要求，或者我们想要一种其他方式的自动代理，比如基于元数据的方式，那么可以在Spring AOP提供的AbstractAutoProxyCreator或者AbstractAdvisorAutoProxyCreator基础之上，实现相应的子类，而不用什么都重新开始。

下面我们看一下Spring AOP框架中有关自动代理的实现架构（见图9-14）。

所有的AutoProxyCreator都是InstantiationAwareBeanPostProcessor，这种类型的BeanPostProcessor与普通的BeanPostProcessor有所不同。当Spring IoC容器检测到有InstantiationAwareBeanPostProcessor类型的BeanPostProcessor的时候，会直接通过InstantiationAwareBeanPostProcessor中的逻辑构造对象实例返回，而不会走正常的对象实例化流程，也就是我说的"短路"。这样，相应的AutoProxyCreator会直接构造目标对象的代理对象返回，而不是原来的目标对象。

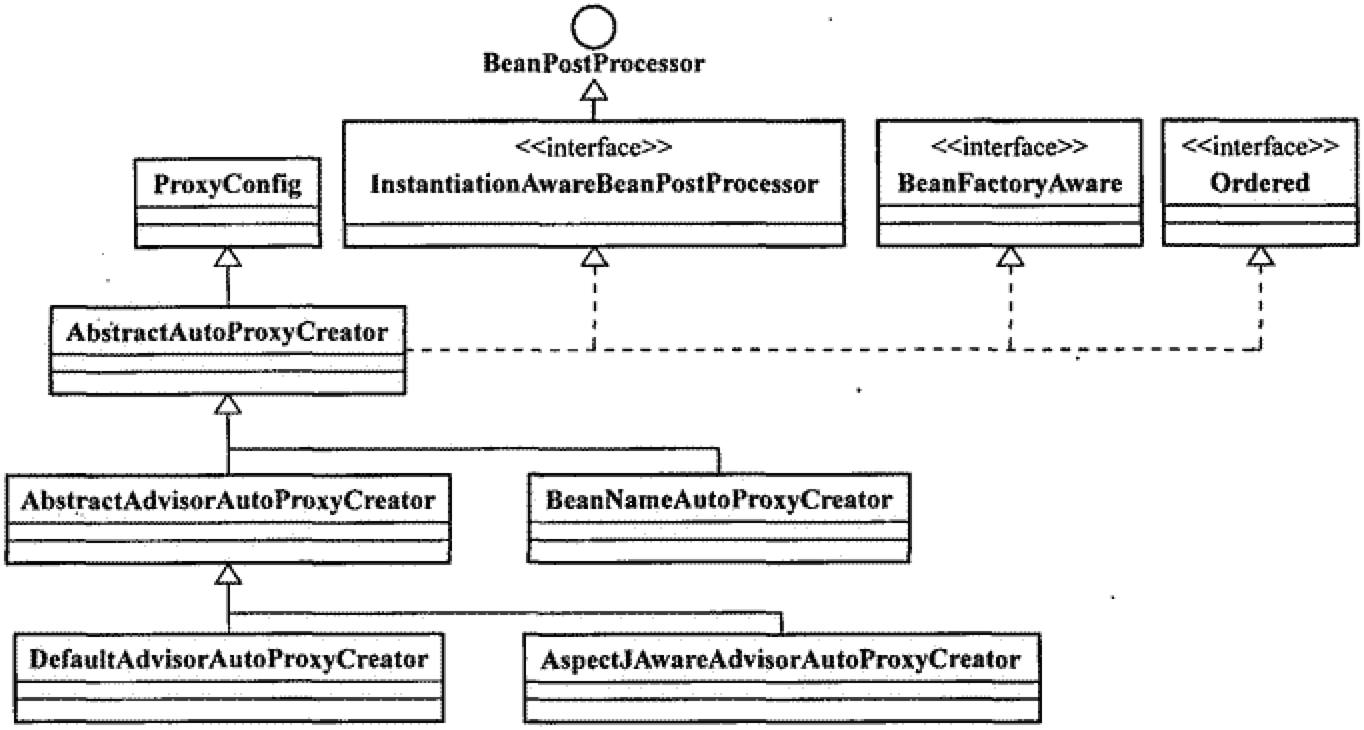

图9-14　AutoProxyCreator实现结构类图

要实现自定义的AutoProxyCreator，可以像BeanNameAutoProxyCreator那样直接继承AbstractAutoProxyCreator，也可以像DefaultAdvisorAutoProxyCreator那样继承AbstractAdvisorAutoProxyCreator。不管如何，子类中只需要提供规则匹配一类的逻辑，如果必要，也可以覆写相应逻辑。

AspectJAwareAdvisorAutoProxyCreator是Spring 2.0之后的AutoProxyCreaotr实现，也算是一个AutoProxyCreator的自定义实现。它还有一个子类AnnotationAwareAspectJAutoProxyCreator，可以支持根据Java 5的注解捕获信息以完成自动代理。我们将在10.1.1节给出更多相关信息。

注意　Spring AOP还支持基于Jakarta Commons Attributes的元数据的自动代理机制。但随着Java 5的普及，Java 5中的注解作为标准将迅速覆盖Java开发领域，所以，基于Jakarta Commons Attribute的自动代理我们就不做解释了，其原理，我们在上面的讲解中也已经提及了，无非是拦截信息的提供方式不同而已。但如果你想使用基于元数据的自动代理，而又不能升级到Java 5，那么可以参照Spring的参考文档，以取得有关在Spring Aop中使用Jakarta Commons Attributes进行自动代理的更多信息。

9.6 TargetSource

通常，在使用ProxyFactory的时候，我们都是通过setTarget()方法指定具体的目标对象。使用ProxyFactoryBean也是如此，或者ProxyFactoryBean还可以通过setTargetName()指定目标对象在IoC容器中的bean定义名称。但除此之外，还有一种方式没有说，那就是还可以通过setTargetSource()来指定目标对象。

TargetSource的作用就好像是为目标对象在外面加了一个壳，或者说，它就像是目标对象的容器。当每个针对目标对象的方法调用经历层层拦截而到达调用链的终点的时候，就该调用目标对象上定义的方法了。但这时，Spring AOP做了点儿手脚，它不是直接调用这个目标对象上的方法，而是通过“插足于”调用链与实际目标对象之间的某个TargetSource来取得具体目标对象，然后再调用从TargetSource中取得的目标对象上的相应方法。整个情形如图9-15所示。

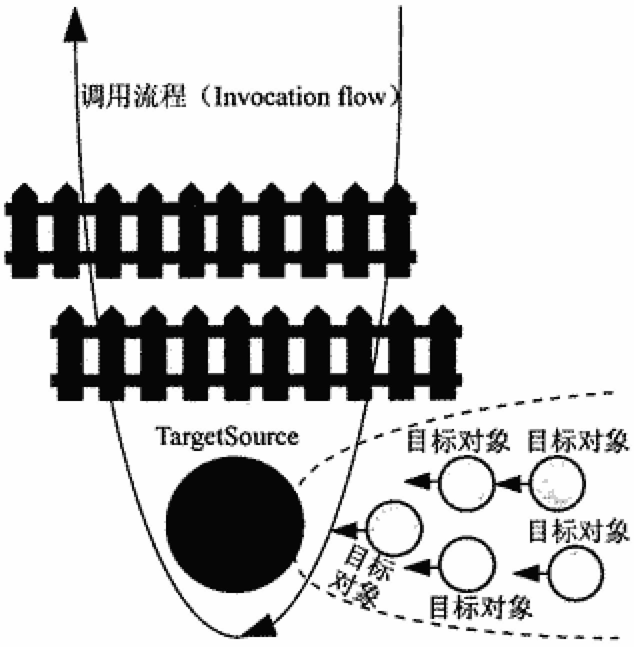

图9-15 TargetSource示意图

在通常情况下，无论是通过setTarget()，还是通过setTargetName()等方法设置的目标对象，框架内部都会通过一个TargetSource实现类对这个设置的目标对象进行封装，也就是说，框架内部会以统一的方式处理调用链终点的目标对象。

TargetSource最主要的特性就是，每次的方法调用都会触发TargetSource的getTarget()方法，getTarget()方法将从相应的TargetSource实现类中取得具体的目标对象，这样，我们就可以控制每次方法调用作用到的具体对象实例：

- 提供一个目标对象池，每次从TargetSource取得的目标对象都从这个目标对象池中取得。
- 让一个TargetSource实现类持有多个目标对象实例，然后按照某种规则，在每次方法调用时，返回相应的目标对象实例。

当然，还可以让TargetSource只持有一个目标对象实例，这样，每次的方法调用就都会针对这一个目标对象实例。其实，这就是通常ProxyFactory或者ProxyFactoryBean处理目标对象的方式，它们内部会构造一个org.springframework.aop.target.SingletonTargetSource实例，而SingletonTargetSource则会针对每次方法调用都返回同一个目标对象实例的引用。

9.6.1 可用的TargetSource实现类

在深入TargetSource的定义之前，我们还是先来看一下有哪些现成的TargetSource实现类。如果这些现有的TargetSource实现类不能满足需求，我们再寻求自定义的TargetSource实现。以下所有的TargetSource实现类全部位于org.springframework.aop.target包中。

1. SingletonTargetSource

org.springframework.aop.target.SingletonTargetSource是使用最多的TargetSource实现类，虽然我们可能并不知道。因为在通过ProxyFactoryBean的setTarget()设置完目标对象之后，ProxyFactoryBean内部会自行使用一个SingletonTargetSource对设置的目标对象进行封装。

SingletonTargetSource的实现其实很简单，就是内部只持有一个目标对象，当每次方法调用到达时，SingletonTargetSource都会返回这同一个目标对象。

所有的TargetSource都可以通过ProxyFactoryBean的setTargetSource()方法进行设置（ProxyFactory同样可以）。为了演示TargetSource的使用，我们来看一个简单的例子（见代码清单9-38）。

代码清单9-38　SingletonTargetSource使用示例

```
<bean id="target" class="..."/>

<bean id="singletonTargetSource" class="org.springframework.aop.target.SingletonTargetSource">
  <constructor-arg>
    <ref bean="target"/>
  </constructor-arg>
</bean>

<bean id="targetProxy" class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="targetSource">
    <ref bean="singletonTargetSource"/>
  </property>
  <property name="interceptorNames">
    <list>
      <value>anyInterceptor</value>
    </list>
  </property>
</bean>
```

该实例其实没有什么太大意义，因为设置一个SingletonTargetSource型的TargetSource，其实跟直接setTarget是等效的。

2. PrototypeTargetSource

与SingletonTargetSource正好相反，如果为ProxyFactory或者ProxyFactoryBean设置一个PrototypeTargetSource类型的TargetSource，那么每次方法调用到达调用链终点，并即将调用目标对象上的方法的时候，PrototypeTargetSource都会返回一个新的目标对象实例供调用。代码清单9-39给出了PrototypeTargetSource的简单使用演示。

代码清单9-39　PrototypeTargetSource使用示例

```
<bean id="target" class="..." singleton="false"/>

<bean id="prototypeTargetSource" class="org.springframework.aop.target.PrototypeTargetSource">
  <property name="targetBeanName">
    <value>target</value>
  </property>
</bean>

<bean id="targetProxy" class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="targetSource">
    <ref bean="prototypeTargetSource"/>
  </property>
  <property name="interceptorNames">
    <list>
      <value>anyInterceptor</value>
    </list>
  </property>
</bean>
```

注意，因为PrototypeTargetSource每次都需要返回新的对象实例，所以，需要注意以下两点。

(1) 目标对象的bean定义声明的scope必须为prototype。

(2) 通过targetBeanName属性指定目标对象的bean定义名称，而不是引用。原因我想就不用多说了，在第4章讲解Spring的IoC容器的时候，我们已经领教过了。

如果对以上配置所取得的targetProxy进行单元测试，那么代码清单9-40所示的测试代码运行之后，应该是可以得到可爱的Green Bar的。（不要告诉我你不知道Green Bar在TDD中意味着什么哦！）

代码清单9-40 针对PrototypeTargetSource的单元测试代码示例

```
...

public void testPrototypeTargetSource() throws Exception
{
  Object proxy = ctx.getBean("targetProxy");
  Object targetObject0 = ((Advised)proxy).getTargetSource().getTarget();
  Object targetObject1 = ((Advised)proxy).getTargetSource().getTarget();
  Object targetObject2 = ((Advised)proxy).getTargetSource().getTarget();
  assertNotSame(targetObject0, targetObject1);
  assertNotSame(targetObject1, targetObject2);
  assertNotSame(targetObject0, targetObject2);
}
```

3. **HotSwappableTargetSource**

这是我觉得比较有用的一个TargetSource实现。使用HotSwappableTargetSource封装目标对象，可以让我们在应用程序运行的时候，根据某种特定条件，动态地替换目标对象类的具体实现，比如，IService有多个实现类，如果程序启动之后，默认的IService实现类出现了问题，我们可以马上切换到IService的另一个实现上，而所有这些对于调用者来说都是透明的。

使用HotSwappableTargetSource的swap方法，可以用新的目标对象实例将旧的目标对象实例替换掉。该方法的声明如下所示：

```
public Object swap(Object newTarget)
```

该方法会返回被替换的旧的目标对象实例。

要使用HotSwappableTargetSource，我们得在它构造的时候，就提供一个默认的目标对象实例，如代码清单9-41所示。

代码清单9-41 HotSwappableTargetSource的初始化

```
<bean id="task" class="org.darrenstudio.books.unveilspring.aop.advisor.MockTask">
</bean>

<bean id="hotSwapTargetSource" class="org.springframework.aop.target.HotSwappableTargetSource">
  <constructor-arg>
    <ref bean="task"/>
  </constructor-arg>
</bean>

<bean id="taskProxy" class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="targetSource" ref="hotSwapTargetSource">
  </property>
  <property name="interceptorNames">
    <list>
      <value>anyInterceptor</value>
    </list>
  </property>
</bean>
```

我们使用构造方法注入为HotSwappableTargetSource注入了初始的目标对象，之后，就可以在程序中对其进行操作，如代码清单9-42所示。

代码清单9-42 HotSwappableTargetSource使用示例

```
...

Object proxy = ctx.getBean("taskProxy");
```

```
Object initTarget = ((Advised)proxy).getTargetSource().getTarget();

HotSwappableTargetSource hotSwapTargetSource = ➥
(HotSwappableTargetSource)ctx.getBean("hotSwapTargetSource");
Object oldTarget = hotSwapTargetSource.swap(new ITask(){
   public void execute(TaskExecutionContext ctx) {
      // 省略
   }});
Object newTarget = ((Advised)proxy).getTargetSource().getTarget();

assertSame(initTarget, oldTarget);
assertNotSame(initTarget,newTarget);
```

能最快记起来的使用HotSwappableTargetSource的场景，就是我曾经在2005年使用它解决了双数据源的互换问题：在有两个数据库双机热备的情况下，如果一个数据库挂掉，则将程序迅速地切换到另一个数据库。当时，使用了ThrowsAdvice对数据库相关异常进行捕捉，在捕捉到必要的切换信息后，就调用HotSwappableTargetSource的swap方法使用新的数据源替换旧的数据源，具体详情可以参照我的博客文章《对双数据源互换的实现的改进》(http://darrenwang.blogcn.com/diary,101689446.shtml)。

总之，使用HotSwappableTargetSource，我们可以在任何合适的地方、合适的时机对旧的目标对象进行替换，比如可以在某个拦截器中持有相关HotSwappableTargetSource的引用，一旦满足相应的条件，就可以调用swap方法动态替换新的目标对象，也可以设置一个定时任务，让它也持有一个HotSwappableTargetSource的引用，每隔一段时间就使用新的目标对象置换掉旧的，等等诸如此类的场景。有关HotSwappableTargetSource的使用，还是有待你自己去挖掘吧！

4. CommonsPoolTargetSource

某些时候，我们可能不想每次都返回新的目标对象，而是想返回有限数目的目标对象实例，这些目标对象实例的“地位”是平等的，就好像数据库连接池中的那些Connection一样，我们可以提供一个目标对象的对象池，然后让某个TargetSource实现每次都从这个目标对象池中去取得目标对象。

CommonsPoolTargetSource就是这么一个TargetSource实现，它使用现有的Jakarta Commons Pool提供对象池支持。使用它，跟使用PrototypeTargetSource没什么太大差别，如代码清单9-43所示。

代码清单9-43　CommonsPoolTargetSource使用配置示例

```
<bean id="target" class="..." singleton="false"/>

<bean id="poolingTargetSource" class="org.springframework.aop.target.CommonsPoolTargetSource">
   <property name="targetBeanName">
      <value>target</value>
   </property>
</bean>

<bean id="targetProxy" class="org.springframework.aop.framework.ProxyFactoryBean">
   <property name="targetSource">
      <ref bean="poolingTargetSource"/>
   </property>
   <property name="interceptorNames">
      <list>
         <value>anyInterceptor</value>
      </list>
   </property>
</bean>
```

在CommonsPoolTargetSource的使用上，需要注意的问题跟PrototypeTargetSource差不多，即注意它要使用prototype型scope的bean定义。CommonsPoolTargetSource还有许多控制对象池的可配

置属性，比如对象池的大小、初始对象数量等，都可以在配置中指定。详情可参照CommonsPool-TargetSource的Javadoc文档。

如果CommonsPoolTargetSource不能满足要求，或者因为其他原因不能使用Jakarta Commons Pool，那么也可以通过扩展org.springframework.aop.target.AbstractPoolingTargetSource类，实现相应的提供对象池化功能的TargetSource。有关扩展AbstractPoolingTargetSource的更多信息，可以参考它的Javadoc或者源码，毕竟Spring是开源的嘛！

5. ThreadLocalTargetSource

如果想为不同的线程调用提供不同的目标对象，那么可以使用org.springframework.aop.target.ThreadLocalTargetSource。它可以保证各自线程上对目标对象的调用，可以被分配到当前线程对应的那个目标对象实例上。其实，ThreadLocalTargetSource无非就是对JDK标准的Thread-Local进行了简单的封装而已。

与其他TargetSource类似，ThreadLocalTargetSource的使用也比较简单，见代码清单9-44。

代码清单9-44 ThreadLocalTargetSource使用配置示例

```
<bean id="target" class="..." singleton="false"/>

<bean id="threadLocalTargetSource" ➥
class="org.springframework.aop.target.ThreadLocalTargetSource">
  <property name="targetBeanName">
    <value>target</value>
  </property>
</bean>

<bean id="targetProxy" class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="targetSource">
    <ref bean="threadLocalTargetSource"/>
  </property>
  <property name="interceptorNames">
    <list>
      <value>anyInterceptor</value>
    </list>
  </property>
</bean>
```

我想你不会忘记将目标对象的bean定义的scope设置成prototype型吧。毕竟人家每个线程起码得有自己的目标对象实例不是？不过没关系，要是非设置成singleton的，Spring AOP会“给你好看”的啦，呵呵，异常伺候！

9

9.6.2 自定义 TargetSource

说了这么多可以使用的TargetSource实现，大部分情况下应该够用了。不过，永远也不能排除特殊情况，我们还得做好实现自定义TargetSource的准备。

要实现自定义的TargetSource，我们可以直接扩展TargetSource接口，好在这个接口定义的方法不多，如下所示：

```
public interface TargetSource extends TargetClassAware {
  Class getTargetClass();
  boolean isStatic();
  Object getTarget() throws Exception;
  void releaseTarget(Object target) throws Exception;
}
```

从下面的方法名称上，我估计各位就能猜出个大概了。

- getTargetClass()方法返回目标对象类型；
- isStatic()用于表明是否要返回同一个目标对象实例，SingletonTargetSource的这个方法肯定是返回true，其他的实现根据情况，通常返回false；
- getTarget()是核心，要返回哪个目标对象实例，完全由它说了算；
- 具体调用过程的结束，如果isStatic()为false，则会调用releaseTarget()以释放当前调用的目标对象。但是否需要释放，完全是由实现的需要决定的，大部分时候，该方法可以空着不实现。

为了演示TargetSource的特性以及如何实现一个TargetSource，我实现了一个简单的AlternativeTargetSource。它内部有一个计数器，当计数器为奇数的时候，TargetSource将针对当前调用返回第一个目标对象实例；否则，返回第二个目标对象实例。AlternativeTargetSource的定义如代码清单9-45所示。

代码清单9-45 AlternativeTargetSource定义

```java
public class AlternativeTargetSource implements TargetSource {
  private ITask alternativeTaskOne;
  private ITask alternativeTaskTwo;

  private int counter;

  public AlternativeTargetSource(ITask task1,ITask task2)
  {
    this.alternativeTaskOne = task1;
    this.alternativeTaskTwo = task2;
  }

  public Object getTarget() throws Exception {
    try
    {
      if(counter %2 == 0)
        return alternativeTaskTwo;
      else
        return alternativeTaskOne;
    }
    finally
    {
      counter++;
    }
  }

  public Class getTargetClass() {
    return ITask.class;
  }

  public boolean isStatic() {
    return false;
  }

  public void releaseTarget(Object arg0) throws Exception {
    // 什么也不做
  }

}
```

在使用AlternativeTargetSource的时候（见代码清单9-46），就会发现，多次的方法调用所发生的目标对象实际上是交错变换的。

代码清单9-46 AlternativeTargetSource的使用示例

```
ITask task1 = new ITask()
{
  public void execute(TaskExecutionContext ctx) {
    System.out.println("execute in Task1.");
  }
};
ITask task2 = new ITask()
{
  public void execute(TaskExecutionContext ctx) {
    System.out.println("execute in Task2.");
  }
};

ProxyFactory pf = new ProxyFactory();
TargetSource targetSource = new AlternativeTargetSource(task1,task2);
pf.setTargetSource(targetSource);
Object proxy = pf.getProxy();
((ITask)proxy).execute(null);
((ITask)proxy).execute(null);
((ITask)proxy).execute(null);
((ITask)proxy).execute(null);
((ITask)proxy).execute(null);
...
```

程序输出，如下所示：

```
execute in Task2.
execute in Task1.
execute in Task2.
execute in Task1.
execute in Task2.
...
```

其实，这个TargetSource实现没有太大的意义，不过扩展一下就说不定了。最主要的，我们可以根据应用的具体场景来给出特定的TargetSource实现。

如果我们的TargetSource自定义实现不在乎是否依赖于Spring的IoC容器，也不妨考虑一下org.springframework.aop.target包中的几个抽象类，直接在这些抽象类的基础上进行扩展，可以省却部分劳烦。

9.7 小结

本章我们详尽剖析了Spring AOP中的各种概念和实现原理，这些概念和实现原理是Spring AOP发布之初就确定的，是整个框架的基础。纵使框架版本如何升级，甚至为Spring AOP加入更多的特性，在升级和加入更多更多特性的过程中，也将一直秉承Spring AOP的这些理念。

了解Spring AOP框架发布之初就确立的各种概念和原理，可以帮助我们更好地理解和使用Spring AOP。甚至，可以帮助我们去扩展Spring AOP。而接下来要讲述的，就是Spring 2.0之后对Spring AOP进行的扩展。

第10章 Spring AOP二世

本章内容

- @AspectJ形式的Spring AOP
- 基于Schema的AOP

10.1 @AspectJ 形式的 Spring AOP

Spring框架2.0版本发布之后，Spring AOP增加了新的特性，或者说增加了新的使用方式。

- 支持AspectJ5发布的@AspectJ形式的AOP实现方式。现在，我们可以直接使用POJO来定义Aspect以及相关的Advice，并使用一套标准的注解标注这些POJO。Spring AOP会根据注解信息查找相关的Aspect定义，并将其声明的横切逻辑织入当前系统。
- 简化了的XML配置方式。现在，使用新的基于XSD的配置方式，我们可以使用aop独有的命名空间，并且注册和使用POJO形式实现的AOP概念实体。因为引入了AspectJ的Pointcut描述语言，也可以在新的配置方式中使用AspectJ形式的Pointcut表达式。

但这只是从使用的角度来看。如果从更“本质”一点儿的角度进行分析的话，我们会发现当升级到2.0版本之后，实际上如下两点是最主要的。

- 可以使用POJO声明Aspect和相关的Advice，而再也不用像1.*x*版本中那样，要实现特定的Advice就需要实现规定的接口。
- 获得了新的Pointcut表述方式，因为现在引入了AspectJ的Pointcut表述语言，再也不用在“直接指定方法名”还是“使用正则表达式”之间选择了。至于说基于XSD的简化的配置方式，应该算是锦上添花之作。

虽然2.0之后的Spring AOP集成了AspectJ，但实际上只能说是仅仅拿来AspectJ的“皮大衣”用一下。而底层各种概念的实现以及织入方式，依然使用的是Spring 1.*x*原先的实现体系。这就好像我们中国人说中国话，而英语在世界上较为普及并且有范围较广的影响力，我们可以学习英语，把英语拿过来为我所用，但本质上，我们还是中国人，而不是英国人。换句话说，Spring AOP还是Spring AOP，只不过多学了门外语而已。

下面让我们看一下当Spring AOP拥有了AspectJ这种表达能力之后，同样的话该怎么来说吧！

@AspectJ代表一种定义Aspect的风格，它让我们能够以POJO的形式定义Aspect，没有其他接口定义限制。唯一需要的，就是使用相应的注解标注这些Aspect定义的POJO类。之后，Spring AOP会根据标注的注解搜索这些Aspect定义类，然后将其织入系统。

这种方式是从AspectJ所引入的，定义的Aspect类基本上可以在Spring AOP和AspectJ之间通用。不过，Spring AOP只使用AspectJ的类库进行Pointcut的解析和匹配，最终的实现机制还是Spring AOP最初

的架构，也就是使用代理模式处理横切逻辑的织入。

下面我们来看看@AspectJ形式是如何使用的！

10.1.1 @AspectJ 形式 AOP 使用之先睹为快

如果将之前的PerformanceMethodInterceptor定义的横切逻辑以@AspectJ形式实现，首先得定义一个Aspect，以最普通的POJO来定义这个Aspect就可以。按照@AspectJ形式重构后的PerformanceMethodInterceptor定义，如代码清单10-1所示。

代码清单10-1 @AspectJ形式的PerformanceMethodInterceptor定义

```
@Aspect
public class PerformanceTraceAspect{
  private final Log logger = LogFactory.getLog(PerformanceTraceAspect.class);

  @Pointcut("execution(public void *.method1()) || execution(public void *.method2())")
  public void pointcutName(){}

  @Around("pointcutName()")
  public Object performanceTrace(ProceedingJoinPoint joinpoint) throws Throwable
  {
    StopWatch watch = new StopWatch();
    try
    {
      watch.start();
      return joinpoint.proceed();
    }
    finally
    {
      watch.stop();
      if(logger.isInfoEnabled()) ➥
        logger.info("PT in method[" ➥
        +joinpoint.getSignature().getName() ➥
        +"]>>>>>"+watch.toString());
    }
  }
}
```

定义这么一个Aspect，我们再也无需像1.x时代的Spring AOP那样实现相应的接口了，现在唯一要做的就是为这个Aspect类加上一个@Aspect的注解。这样，稍后我们可以根据这个@Aspect，来判断Classpath中哪些类是我们要找的Aspect定义。

我们知道，Aspect中可以定义多个Pointcut以及多个Advice，所以，除了要使用@Aspect标注Aspect类之外，还需要通过名为@Pointcut的注解指定Pointcut定义，通过@Around等注解来指定哪些方法定义了相应的Advice逻辑。至于说这些注解如何使用，以及对应的方法定义还有什么需要注意的地方，我们先不要管，稍后会详细讲述。

假设我们有如下目标对象类定义：

```
public class Foo {
  public void method1(){
    System.out.println("method1 execution.");
  }
  public void method2(){
    System.out.println("method2 execution.");
  }
}
```

现在有两种方式将Aspect定义织入这个目标对象类，实现对其符合Pointcut定义的Joinpoint（也就是方法执行）进行拦截。

1. 编程方式织入

还记得在讲解ProxyFactory的时候，除了ProxyFactoryBean，我们还提到ProxyFactory的另一个"兄弟"吗？对，那就是org.springframework.aop.aspectj.annotation.AspectJProxyFactory。

通过AspectJProxyFactory，我们就可以实现Aspect定义到目标对象的织入，这样就有了如下代码所示的编程方式织入过程：

```
AspectJProxyFactory weaver = new AspectJProxyFactory();
weaver.setProxyTargetClass(true);
weaver.setTarget(new Foo());
weaver.addAspect(PerformanceTraceAspect.class);
Object proxy = weaver.getProxy();
((Foo)proxy).method1();
((Foo)proxy).method2();
```

AspectJProxyFactory的使用与ProxyFactory没有多大差别，只不过多了addAspect()方法，通过该方法可以直接为AspectJProxyFactory添加相应的Aspect定义。实际上，如果我们愿意，完全可以把AspectJProxyFactory当作ProxyFactory来用！

2. 通过自动代理织入

针对@AspectJ风格的AOP，Spring AOP专门提供了一个AutoProxyCreator实现类进行自动代理，以免去过多编码和配置的工作，这个AutoProxyCreator我们之前也提到过，即org.springframework.aop.aspectj.annotation.AnnotationAwareAspectJAutoProxyCreator。它是在AbstractAdvisorAutoProxyCreator的基础上给出的一个扩展类，它的直接父类是AspectJAwareAdvisorAutoProxyCreator。

与使用其他AutoProxyCreator一样，我们只需要在IoC容器的配置文件中注册一下AnnotationAwareAspectJAutoProxyCreator就可以了，如下配置所示：

```
<bean class="org.springframework.aop.aspectj.annotation. ➥
AnnotationAwareAspectJAutoProxyCreator">
   <property name="proxyTargetClass" value="true"></property>
</bean>

<bean id="performanceAspect" class="org.darrenstudio.books. ➥
unveilspring.aop.aspectj.PerformanceTraceAspect"/>

<bean id="target" class="...Foo">
</bean>
```

现在，AnnotationAwareAspectJAutoProxyCreator会自动搜集IoC容器中注册的Aspect，并应用到Pointcut定义的各个目标对象上。如果我们通过容器取得现在的target对象的话，会发现它已经是被代理过的了，如下：

```
ApplicationContext ctx = new ClassPathXmlApplicationContext("...");
Object proxy = ctx.getBean("target");
((Foo)proxy).method1();
((Foo)proxy).method2();
```

当然，如果把target作为依赖对象注入其他的bean定义，那么依赖的主体对象现在持有的也是被代理过的目标对象。

刚才AnnotationAwareAspectJAutoProxyCreator注册到容器的方式是基于DTD的配置方式，在Spring 1.x以及2.x版本中都可以使用。如果我们能够使用Spring 2.x版本，并且使用基于XSD的配置方式，还可以有另一种更加简洁的配置方式，如代码清单10-2所示。

代码清单10-2 基于XSD的AspectJ形式的自动代理功能启用的配置示例

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" ➥
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ➥
xmlns:aop="http://www.springframework.org/schema/aop" ➥
xsi:schemaLocation="http://www.springframework.org/schema/beans ➥
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd ➥
http://www.springframework.org/schema/aop ➥
http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

    <aop:aspectj-autoproxy proxy-target-class="true">
    </aop:aspectj-autoproxy>

    <bean id="target" class="org.darrenstudio.books.unveilspring.aop.aspectj.Foo">
    </bean>

    <bean id="performanceAspect" class="org.darrenstudio.books.unveilspring.aop.aspectj. ➥
    PerformanceTraceAspect"/>

</beans>
```

通过面向aop命名空间的<aop:aspectj-autoproxy>，可以达到与基于DTD的配置方式中，直接声明AnnotationAwareAspectJAutoProxyCreator相同的效果。该元素背后的工作实际上就是由AnnotationAwareAspectJAutoProxyCreator来做的。另外，不要忘了将aop命名空间的Schema定义引入XSD定义。我想，要用<aop:aspectj-autoproxy>的话，应该也不会忘记这一点！

> 小心 在使用@AspectJ形式的AOP的时候，应该尽量使用容器内的自动代理支持。通常，只有出于测试目的，才会使用编程的方式进行直接的织入操作。在使用的过程中，你会发现，实际上这两种织入方式是有差异的，一些行为并不统一。

从现在开始，只要使用到了@Aspect形式的AOP，就不要忘记将aspectjweaver.jar和aspectjrt.jar加入到应用程序的Classpath中。因为有些具体工作是要求助于AspectJ的类库的，要知道都哪些具体任务，我们接着往下看吧！

10.1.2 @AspectJ 形式的 Pointcut

在Spring框架发布2.0版本之前，Spring AOP没有像AspectJ那样的正式的Pointcut描述语言，而且也只支持方法级别的拦截。所以通常情况下，简单的方法名指定以及正则表达式两种方式，基本可以很好地达到目的。在Spring发布2.0版本之后，Spring AOP框架集成了AspectJ的部分功能，这其中就包括AspectJ的Pointcut描述语言支持。

1. @AspectJ形式Pointcut的声明方式

@AspectJ形式的Pointcut声明，依附在@Aspect所标注的Aspect定义类之内，通过使用org.aspectj.lang.annotation.Pointcut这个注解，指定AspectJ形式的Poincut表达式之后，将这个指定了相应表达式的注解标注到Aspect定义类的某个方法上即可。代码清单10-3给出了一个使用@Pointcut的简单示例。

代码清单10-3 使用@Pointcut声明相应Pointcut的示例

```
@Aspect
public class YourAspect
{
   @Pointcut("aspectj style pointcut expression")  // pointcut_expression
   public void pointcutMethod(){}     // pontcut_signature

   @Pointcut("aspectj style pointcut expression") // pointcut_expression
   private void pointcutMethod2(){}  // pontcut_signature

   ...
}
```

@AspectJ形式的Pointcut声明包含如下两个部分。

- Pointcut Expression。Pointcut Expression的载体为`@Pointcut`，该注解是方法级别的注解，所以，Pointcut Expression不能脱离某个方法单独声明。Pointcut Expression附着于上的方法称为该Pointcut Expression的Pointcut Signature。Pointcut Expression是真正规定Pointcut匹配规则的地方，可以通过@Pointcut直接指定AspectJ形式的Pointcut表达式。@Pointcut所指定的AspectJ形式的Pointcut表达式由如下两部分组成。
  - Pointcut标志符（Pointcut Designator）。标志符表明该Pointcut将以什么样的行为来匹配表达式，可以使用的Pointcut表达式将在后文中详细讲述。
  - 表达式匹配模式。在Pointcut标志符之内可以指定具体的匹配模式，模式的规则将在本节的第2小节中提及。
- Pointcut Signature。Pointcut Signature在这里具体化为一个方法定义，它是Pointcut Expression的载体。Pointcut Signature所在的方法定义，除了返回类型必须是`void`之外，没有其他限制。方法修饰符所起的作用与Java语言中语义相同，`public`型的Pointcut Signature可以在其他Aspect定义中引用，`private`则只能在当前Aspect定义中引用。可以将Pointcut Signature作为相应Pointcut Expression的标志符，在Pointcut Expression的定义中取代重复的Pointcut表达式定义，如代码清单10-4所示。

代码清单10-4 在Pointcut Expression中使用Pointcut Signature的示例

```
@Aspect
public class YourAspect
{
   @Pointcut("execution(void method1())")  // pointcut_expression
   public void method1Execution(){}     // pointcut_signature

   @Pointcut("method1Execution()") // pointcut_expression
   private void stillMehtod1Execution(){}  // pointcut_signature

   ...
}
```

`stillMehtod1Execution()`的Pointcut Expression，通过第一个Pointcut定义的Pointcut Signature，即`method1Execution()`，引用了第一个Pointcut定义，所以，这两个Pointcut定义的规则是一样的。

AspectJ的Pointcut表达式支持通过`&&`、`||`以及`!`逻辑运算符，进行Pointcut表达式之间的逻辑运算，运算符可以应用于具体的Pointcut表达式，以及相应的Pointcut Signature。代码清单10-5演示了这几种逻辑运算符的使用。

代码清单10-5 @Pointcut中的逻辑运算示例演示

```
@Aspect
public class YourAspect
{
   @Pointcut("execution(void method1())")
   private void method1Exec(){}

   @Pointcut("execution(void method2())")
   private void method2Exec(){}

   @Pointcut("execution(void method1()) || execution(void method2())")
   public void bothMethodExec(){}

   @Pointcut("method1Exec() || method2Exec()")
   public void bothMethodExec2(){}

   ...
}
```

可以看到，通过前两个简单的Pointcut定义以及相应的逻辑运算，可以得到更为复杂的Pointcut定义。因为前两个简单的Pointcut定义只需要在当前Aspect内引用，所以，我们声明为`private`。不过，我们也完全可以将其声明为`public`型的，这样，其他的Aspect中的Pointcut定义也同样可以引用的到。对于一个系统中能够公用或者统一管理的一类Pointcut来说，完全可以声明一个专门的Aspect来定义这些Pointcut，如代码清单10-6所示。

代码清单10-6 包含公用Pointcut声明的SystemCommonsAspect定义

```
@Aspect
public class SystemCommonsAspect
{
   @Pointcut("pointcut expression")
   public void commonPointcut1(){}

   @Pointcut("pointcut expression")
   public void commonPointcut2(){}

   ...
}
```

然后，在其他Aspect定义的Pointcut定义中引用它们，以避免重复定义，比如：

```
@Aspect
public class OtherAspect
{
   @Pointcut("package.SystemCommonsAspect.commonPointcut1() || ➥
   package.SystemCommonsAspect.commonPointcut2()")
   public void compostePOintcutDefinition()

   ...
}
```

为了能够让Pointcut表达式能够运行起来，我们还是抓紧时间来看一下都有哪些可以使用的Pointcut标志符吧！

2. @AspectJ形式Pointcut表达式的标志符

虽然AspectJ的Pointcut表达式可用的标志符很丰富，基本上可以囊括所有Joinpoint类型的表述，但是，因为Spring AOP只支持方法级别的Joinpoint，所以可以通过`AspectJExpressionPointcut`指定的

AspectJ形式的Pointcut表达式是有一定限制的，我们只能使用AspectJ的Pointcut表述语言中的少数几种标志符。

- execution

Spring AOP仅支持方法执行类型的Joinpoint，所以execution将会是我们使用最多的标志符，使用它，将帮助我们匹配拥有指定方法签名的Joinpoint。使用execution标志符的Pointcut表达式的规定格式如下：

```
execution(modifiers-pattern? ret-type-pattern declaring-type-pattern? ➥
name-pattern(param-pattern) throws-pattern?)
```

其中，方法的返回类型、方法名以及参数部分的匹配模式是必须指定的，其他部分的匹配模式可以省略。

假设我们拥有以下类定义：

```
public class Foo{
  public void doSomething(String arg){
    ...
  }
}
```

那么可以使用如下的Pointcut表达式来匹配Foo的doSomething的方法执行：

```
execution(public void Foo.doSomething(String))
```

因为部分匹配模式可以省略，所以，我们也可以简化以上模式，如下所示：

```
execution(void doSomething(String))
```

除此之外，我们还可以在execution的表达式中使用两种通配符，即*和..。

- *可以用于任何部分的匹配模式中，可以匹配相邻的多个字符，即一个Word。使用*之后，我们以上的execution表达式就可以简化成：

  ```
  execution(* *(String))
  ```

  当然这样的简化之后，匹配的范围要比原来的表达式所匹配的范围要广得多。另外，我们还可以在方法参数匹配模式中也使用*来匹配，如下所示：

  ```
  execution(* *(*))
  ```

  在这里，以上表达式表示只有一个参数的方法，参数类型可以为任何类型。

- ..通配符可以在两个位置使用，一个是在declaring-type-pattern规定的位置，一个在方法参数匹配模式的位置。如果用于declaring-type-pattern规定的位置，则可以指定多个层次的类型声明，如下：

  ```
  execution(void cn.spring21.*.doSomething(*))
  // 只能指定到cn.spring21这一层下的所有类型

  execution(void cn.spring21..*.doSomething(*))
  // 可以匹配cn.spring21包下的所有类型，以及cn.spring21下层包下声明的所有类型
  ```

  如果..用于方法参数列表匹配位置，则表示该方法可以有0到多个参数，参数类型不限，如下：

  ```
  execution(void *.doSomething(..))
  ```

注意，如果在这里使用*，则只能匹配一个参数。另外，我们还可以进行*、..与具体参数类型的组合，如下所示：

```
execution(void doSomething(String,*)
// 匹配两个参数的doSomething方法，第一个参数为String类型，第二个参数类型不限

execution(void doSomething(..,String)
// 匹配拥有多个参数的doSomething方法，之前几个参数类型不限，但最后一个参数类型必须为String

execution(void doSomething(*,String,..)
// 匹配拥有多个参数的doSomething方法，第一个参数类型不限，第二个参数类型为String，
// 其他剩余参数数量，类型均不限
```

好啦，就说这么些吧，更多能量还是由你来发掘吧。

- within

within标志符只接受类型声明，它将会匹配指定类型下所有的Joinpoint。不过，因为SpringAOP只支持方法级别的Joinpoint，所以，在我们为within指定某个类后，它将匹配指定类所声明的所有方法执行。假设我们声明一个使用within标志符的Pointcut如下：

```
within(cn.spring21.aop.target.MockTarget)
```

那么，该Pointcut表达式在Spring AOP中将会匹配MockTarget类中的所有方法声明。另外，我们也可以通过*和..通配符来扩展匹配的类型范围，如下所示：

```
within(cn.spring21.aop.target.*)
// 匹配cn.spring21.aop.target包下所有类型内部的方法级别的Joinpoint

within(cn.spring21.aop..*)
// 匹配cn.spring21.aop.target以及其子包下所有类型的内部方法级别的Joinpoint
```

- this和target

之所以将这两个标志符放在一起讲，是因为通过对它们的对比，可以更清楚地了解这两个标志符各自的语义。

在AspectJ中，this指代调用方法一方所在的对象（caller），target指代被调用方法所在的对象（callee），这样通常可以同时使用这两个标志符限定方法的调用关系。比如，如果Object1、Object2都会调用Object3的某个方法，那么，Pointcut表达式定义this(Object2) && target(Object3)只会当Object2调用Object3上的方法的时候才会匹配，而Object1调用Object3上的方法则不会被匹配。

Spring AOP中的this和target标志符语义，有别于AspectJ中这两个标志符的原始语义。现在，this指代的是目标对象的代理对象，而target如其名般指代的就是目标对象。如果使用this(ObjectType)作为Pointcut定义，那么当目标对象的代理对象是ObjectType类型的时候，该Pointcut定义将匹配ObjectType类型中所有的Joinpoint。在Spring AOP中，也就是ObjectType中定义的所有方法。而使用target(ObjectType)作为Pointcut定义，当目标对象是ObjectType类型的时候，该Pointcut定义将匹配ObjectType型目标对象内定义的所有Joinpoint。在Spring AOP中，当然还是所有的方法执行。

实际上，从代理模式来看，代理对象通常跟目标对象的类型是相同的，因为目标对象与它的代理对象实现同一个接口。即使使用CGLIB的方式，目标对象的代理对象属于目标对象的子类型，通过单独使用this或者target指定类型，起到的限定作用其实是差不多的。假设我们有对象定义，如下：

```
public class TargetFoo implements ProxyInterface{
   ...
}

public interface ProxyInterface{
```

```
    ...
}
```

不论使用基于接口的代理方式，还是基于类的代理方式，如下两个Pointcut表达式所起的所用实际上是差不多的：

```
this(ProxyInterface)

target(ProxyInterface)
```

因为TargetFoo作为目标对象实现了ProxyInterface。对基于接口的代理来说，它的代理对象同样实现了这个接口。对于基于类的代理来说，因为目标对象的代理对象是继承了目标对象，自然也继承了目标对象实现的接口。所以，在这里，这两个Pointcut定义起得作用差不多。如果通过this和targe指定具体类型，会怎么样呢？如下所示：

```
this(TargetFoo)

target(TargetFoo)
```

这时，对于基于接口的代理和基于类的代理来说，效果就不同了。对于前者来说，target-(TargetFoo)可以匹配目标对象中的所有Joinpoint，因为目标对象确实是TargetFoo类型，而this(TargetFoo)则不可以。此时，这两个标志符出现分歧了。不过，对于后者，即基于类的代理来说，这两个Pointcut表达式限定的语义还是基本相同的。

通常，this和target标志符都是在Pointcut表达式中与其他标志符结合使用，以进一步加强匹配的限定规则，比如：

```
execution(void cn.spring21.*.doSomething(*)) && this(TargetFoo)
或者
execution(void cn.spring21.*.doSomething(*)) && target(ProxyInterface)
```

当然，我们也可以将this和target结合到一个Pointcut定义中。这样，在目标对象和代理对象关注的类型不同的时候，可以达到严格匹配规则的目的。为了说明这一点，让我们的TargetFoo定义再多实现一个接口吧！代码清单10-7给出了变更后的相关类定义。

代码清单10-7　TargetFoo及相关接口定义

```
public class TargetFoo implements ProxyInterface,ProxyInterface2{
    ...
}

public interface ProxyInterface{
    ...
}

public interface ProxyInterface2{
    ...
}
```

现在，我们定义Pointcut，如下所示：

```
this(ProxyInterface) && target(ProxyInterface2)
```

当为目标对象生成代理对象时，我们声明只对ProxyInterface接口进行代理，那么使用以上Pointcut表达式可以匹配TargetFoo。如果还有其他的ProxyInterface实现类，但是它们没有同时实现ProxyInterface2，那么这些其他的ProxyInterface实现类则不会被匹配。

除了以上这种代理对象实现的接口比目标对象少的情况，对于Introduction来说，代理对象所实现的接口数量通常比目标对象多，也同样可以同时使用this和target进一步限定匹配的规则，比如：

```
this(IntroductionInterface) && target(TargetObjectType)
```

Introduction为目标对象动态添加了新的接口，但是新的接口添加到了目标对象的代理对象上，所以，在这一点上，this和target可以指定的特性是有差异的。

总之，在理解this和target的匹配行为的基础上，应该尽量将this和target标志符与其他标志符一起使用。

- args

该标志符的作用是，帮助我们捕捉拥有指定参数类型、指定参数数量的方法级Joinpoint，而不管该方法在什么类型中被声明。

如果我们声明Pointcut表达式如下：

```
args(cn.spring21.unveilspring.domain.User)
```

那么，如下的方法签名都将被该Pointcut所匹配：

```
public class Foo{
  public boolean login(User user){...};
}

public class Bar{
  public boolean isLogin(User user){...};
}
```

但是，与使用execution标志符可以直接明确指定方法参数类型不同，args标志符会在运行期间动态检查参数的类型，即使我们的方法签名声明如下：

```
public boolean login(Object user);
```

只要传入的是User类型的实例，那么使用args标志符的Pointcut表达式依然可以捕捉到该Joinpoint。但是，类似于execution(* *(User))这样的Pointcut表达式，则无法捕捉以上Pointcut形式声明的Joinpoint，因为它是静态的Pointcut。

- @within

如果使用@within指定了某种类型的注解，那么，只要对象标注了该类型的注解，使用了@within标志符的Pointcut表达式将匹配该对象内部所有Joinpoint。对于Spring AOP来说，当然是对象内部声明的所有方法级Joinpoint。例如，假设我们声明注解如下：

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD,ElementType.TYPE})
public @interface AnyJoinpontAnnotation {

}
```

并且，拥有类声明如下：

```
@AnyJoinpontAnnotation
public class Foo{
  public void method1(){...}
  public void method2(){...}
  ...
}
```

当使用@within标志符声明Pointcut表达式时，如下所示：

```
@within(AnyJoinpontAnnotation)
```

Foo类中的method1、method2等方法将全部被该Pointcut表达式所匹配。因为该类标注了@AnyJoinpontAnnotation，其他类如果也标注了该注解，也会被一概纳入囊中。

> 小心　@within只接受注解类型，并对被指定注解类型所标注的类生效。

- @target

如果目标对象拥有@target标志符所指定的注解类型，那么目标对象内部所有的Joinpoint将被匹配。当然，对于Spring AOP来说，是目标对象中所有的方法级别Joinpoint将被匹配。

以@within部分使用的Foo类和它相关的@AnyJoinpontAnnotation为例，如果指定Pointcut表达式如下：

```
@target(AnyJoinpontAnnotation)
```

只要当前的目标对象像Foo那样标注了@AnyJoinpontAnnotation，则目标对象内部的所有方法级别Joinpoint将被匹配。

在Spring AOP中，@within和@target没有太大的区别。只不过@within属于静态匹配，而@target则是在运行时点动态匹配Joinpoint。

- @args

使用@args标志符的Pointcut表达式将会尝试检查当前方法级的Joinpoint的方法参数类型。如果该次传入的参数类型拥有@args所指定的注解，当前Joinpoint将被匹配，否则将不会被匹配。如图10-1所示。

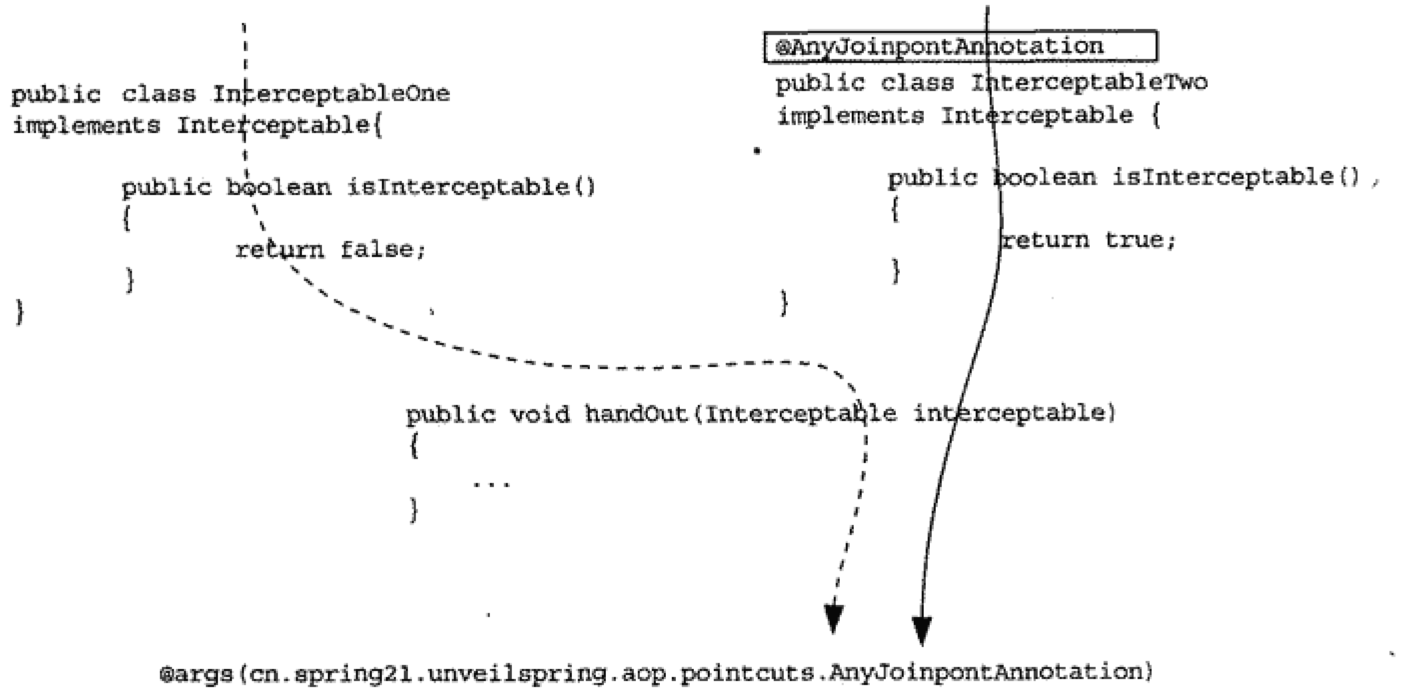

图10-1　@args形式Pointcut的匹配示意图

第一次方法调用，传入的参数为InterceptableOne类型的实例，而该类型没有标注指定的注解，即@AnyJoinpontAnnotation，所以当前handOut方法执行将不会被Pointcut表达式所匹配。第二次方

法调用，传入的参数为InterceptableTwo，而它标注了@AnyJoinpontAnnotation，所以，这次方法执行将被匹配。

@args会尝试对系统中所有对象的每次方法执行的参数，都进行指定的注解的动态检查。只要参数类型标注有@args指定的注解类型，当前方法执行就将匹配，至于说参数类型是什么，它则不是十分关心。

> 小心 @args只接受注解类型声明！另外，我尝试在同一个@args标志符中同时指定两个注解，类似于@args(annotationType1,annotationType2)。但Spring AOP不予理睬。呵呵，不过没关系，稍后使用逻辑运算符就可以了！

- @annotation

使用@annotation标志符的Pointcut表达式，将会尝试检查系统中所有对象的所有方法级别Joinpoint。如果被检测的方法标注有@annotation标志符所指定的注解类型，那么当前方法所在的Joinpoint将被Pointcut表达式所匹配。

@annotation标志符的应用场景也比较广泛，尤其是使用Java 5之后。假设我们要对系统中的事务处理进行统一管理，除了稍后将介绍的声明性事务处理之外，我们也可以通过团队内部规定的使用注解的方式管理事务。当开发人员希望对某个方法加事务控制的时候，只要使用相应的注解标注一下即可，如下所示：

```
public class MockService{

   @Transactional
   public void service(){
      ...
   }
}
```

之后，我们只要通过以下Pointcut表达式，就可以匹配所有这些标注了@Transactional的方法，然后为其织入事务控制逻辑：

```
@annotation(org.springframework.transaction.annotation.Transactional)
```

要是早有这个东西就好了，也省得我在之前的项目中重新实现了一套类似的功能。

最终需要注意的就是，@annotation所接受的注解类型只应用于方法级别，即标注了@Target(ElementType.METHOD)的注解声明。

> 注意 所有以@开头的标志符，都只能指定注解类型参数，而且，注解只有Java 5甚至之后才有效。所以，你已经猜到了，要用这些，就得祈祷我们的系统已经升级到Java 5或者更高版本的JVM了。

在Spring AOP中，如果要使用AspectJ的Pointcut表达式来指定Pointcut，只允许使用以上列出的几种标志符。AspectJ中同样可以使用的initialization、get、set、handler等标志符，在Spring AOP中是不支持的。如果通过@Pointcut设置了包含这些不支持的标志符的Pointcut表达式，Spring AOP将抛出IllegalArgumentException而拒绝这种“无礼请求”。毕竟，Spring AOP只是借用一下AspectJ的Pointcut表述语言，而底层的Joinpoint类型匹配却依然是Spring最初的承诺。

即使是在Spring AOP中可以使用的AspectJ中的Pointcut表达式标志符，它们的语义跟AspectJ中的

最初语义也会或多或少有所偏差。所以，与其说Spring AOP现在可以使用AspectJ的Pointcut表述语言，不如说Spring AOP借用了AspectJ的Pointcut表述语言的“外衣”，而实际上底层的语意和最终匹配，却还是沿用Spring AOP最初的机制。另外，将来Spring AOP还可能在原来表达式的基础上增加新的标志符，如`bean(...)`[①]。而这种bean标志符在AspectJ的Pointcut表述语言中是不存在的，AspectJ扩展了Java语言，现在Spring AOP的Pointcut表达式则要扩展AspectJ的Pointcut表达式了。

> **注意**　可以参照The AspectJ Programming Guide（http://www.eclipse.org/aspectj/doc/released/progguide/index.html）获得有关Pointcut表达式以及相关标志符的详细信息。

3. @AspectJ形式的Pointcut在Spring AOP中的真实面目

实际上，@AspectJ形式声明的所有Pointcut表达式，在Spring AOP内部都会通过解析，转化为具体的Pointcut对象。因为Spring AOP有自己的Pointcut定义结构，所以，@AspectJ形式声明的这些Pointcut表达式，最终会转化成一个专门面向AspectJ的Pointcut实现。

`org.springframework.aop.aspectj.AspectJExpressionPointcut`代表Spring AOP中面向AspectJ的Pointcut具体实现。虽然它会使用AspectJ的相应支持，但依然遵循Spring AOP的Pointcut定义，其在Pointcut中的地位如图10-2所示。

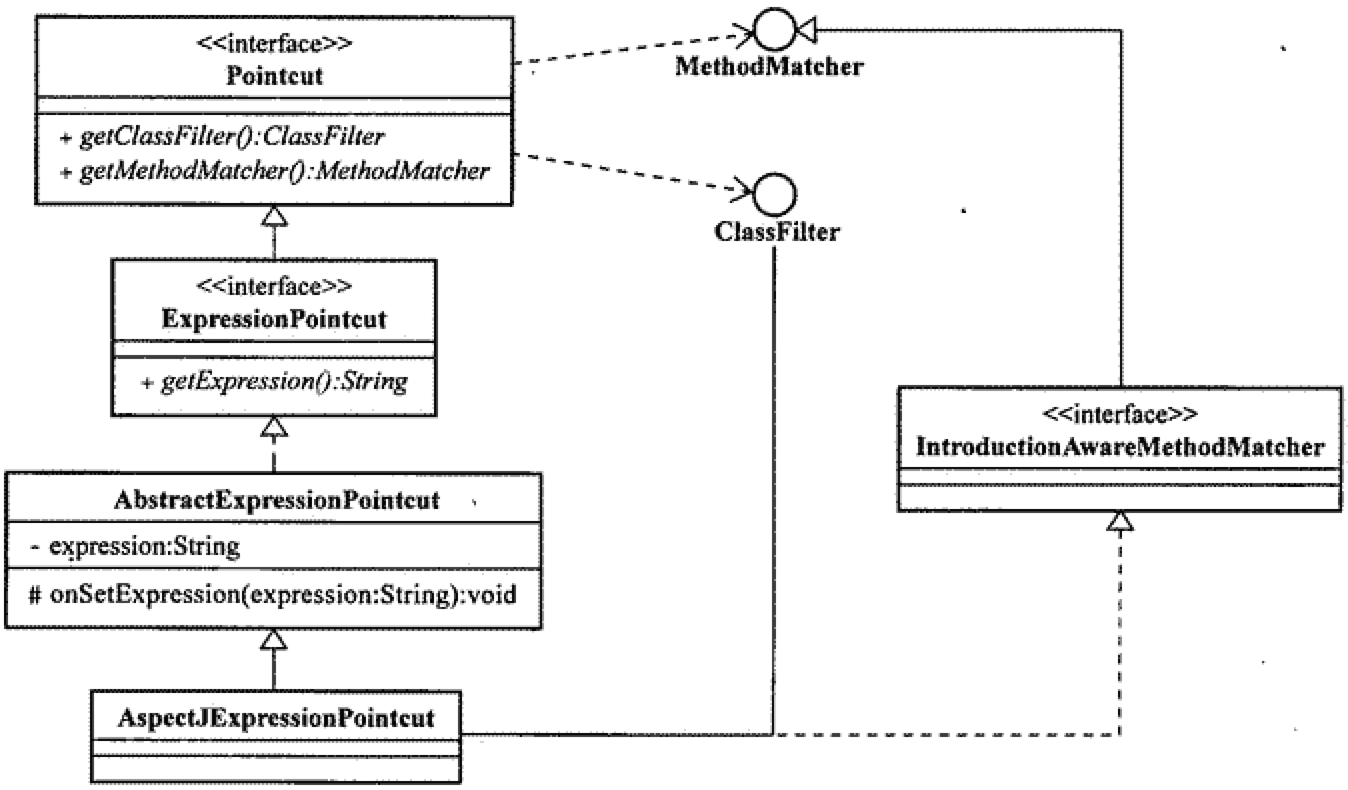

图10-2　AspectJ Pointcut扩展的类图

定义`ExpressionPointcut`和`AbstractExpressionPointcut`主要是为了以后的扩展性。如果还有AspectJ的Pointcut描述语言之外的形式，我们可以在这两个的基础上进行集成。

---

① Spring 2.5中已经对bean标志符提供了正式的支持，详细情况可以参见Spring 2.5的参考文档，或者来自Spring开发团队博客上的一篇博文“The new bean() pointcut”，其对bean标志符做了生动的介绍。可以从http://blog.springsource.com/main/2007/09/24/the-new-bean-pointcut/网址访问到这篇文字。

在AspectJProxyFactory或者AnnotationAwareAspectJAutoProxyCreator通过反射获取了Aspect中的@Pointcut定义的AspectJ形式的Pointcut定义之后，在Spring AOP框架内部都会构造一个对应的AspectJExpressionPointcut对象实例。AspectJExpressionPointcut内部持有通过反射获得的Pointcut表达式。

AspectJExpressionPointcut属于Spring AOP的Pointcut定义之一，所以，Spring AOP框架内部处理Pointcut匹配的逻辑不需要改变，依然使用原来的匹配机制，即通过ClassFilter和MethodMatcher进行具体Joinpoint的匹配工作。不过，AspectJExpressionPointcut在实现ClassFilter和MethodMatcher相应方法逻辑的时候，会委托AspectJ类库的相关类来做具体的工作。AspectJExpressionPointcut会委托AspectJ类库中的PointcutParser来对它所持有的AspectJ形式的Pointcut表达式进行解析。PointcutParser解析完成之后会返回一个PointcutExpression对象（依然是AspectJ类库中的类），之后，匹配与否就直接委托这个PointcutExpression对象的相关方法进行处理了。

AspectJExpressionPointcut属于面向AspectJ的Pointcut实现，所以，我们可以像Spring AOP中其他各种Pointcut实现类那样来使用它。只不过，构造完成后，我们不是为它设置要拦截的方法名或者正则表达式，而是如下所示设置@AspectJ形式的Pointcut表达式：

```
AspectJExpressionPointcut pointcut = new AspectJExpressionPointcut();
pointcut.setExpression("execution(* someMethodName(..))");
// 可以使用状态
```

当然，我们使用@AspectJ形式的AOP是没有必要通过编码来使用AspectJExpressionPointcut的，直接在Aspect中通过@Pointcut指定AspectJ形式的Pointcut表达式，比编码使用AspectJ-ExpressionPointcut要简洁得多。不过，如果愿意，我们完全可以在1.x版本的Spring AOP中使用AspectJExpressionPointcut类型的Pointcut来指定具体的Pointcut定义，并不是说AspectJEx-pressionPointcut只有在@Aspect形式的AOP中才可以使用。

10.1.3 @AspectJ 形式的 Advice

@AspectJ形式的Advice定义，实际上就是使用@Aspect标注的Aspect定义类中的普通方法。只不过，这些方法需要针对不同的Advice类型使用对应的注解进行标注。

可以用于标注对应Advice定义方法的注解包括：

- @Before。用于标注Before Advice定义所在的方法；
- @AfterReturning。用于标注After Returning Advice定义所在的方法；
- @AfterThrowing。用于标注After Throwing Advice定义所在的方法，也就是在Spring AOP中称为ThrowsAdvice的那种Advice类型；
- @After。用于标注After(finally) Advice定义所在的方法，1.x版本的Spring AOP中没有对应这种类型的Advice接口定义或者实现；
- @Around。用于标注Around Advice定义所在的方法，也就是常说的拦截器类型的Advice；
- @DeclareParents。用于标注Introduction类型的Advice，但该注解对应标注对象的域（Field），而不是方法（Method）。

除了@DeclareParents比较不同之外，其他用于标注不同类型Advice的注解，全部都是方法级别的注解定义，只能用于标注方法定义。同时，各种Advice最终织入到什么位置，是由相应的Pointcut定义决定的。所以，我们需要为这些用于标注Advice的注解指定对应的Pointcut定义，可以直接指定

@AspectJ形式的Pointcut表达式，也可以指定单独声明的@Pointcut类型定义的Pointcut Signature（不包括 @DeclareParents，它的使用会单独详细阐述），如代码清单10-8所示。

代码清单10-8 各种Advice对应的注解使用示例

```
@Aspect
public class MockAspect
{
   @Pointcut("execution(* destroy(..)")
   public void destroy(){}

   @Before("execution(public void *.methodName(String))")
   public void setUpResourceFolder()
   {
      ...
   }

   @After("destroy()")
   public void cleanUpResourcesIfNecessary()
   {
      ...
   }
   ...
}
```

好了，还是让我们详细看一下各种Advice在@AspectJ形式中的使用方式和可能需要注意的问题吧！

1. Before Advice

以@AspectJ形式声明Before Advice，只需要在Aspect定义类中定义相应的方法，然后使用org.aspectj.lang.annotation.Before标注该方法即可。

如果将之前的ResourceSetupBeforeAdvice定义移植到@AspectJ形式，我们可以如代码清单10-9所示定义其横切逻辑。

代码清单10-9 @AspectJ形式的Before Advice定义示例

```
@Aspect
public class ResourceSetupJAspect {
   private Resource resource;

   @Before("execution(boolean *.execute())")
   public void setupResourcesBefore() throws Throwable
   {
      if(!getResource().exists())
      {
         FileUtils.forceMkdir(getResource().getFile());
      }
   }

   public Resource getResource() {
      return resource;
   }

   public void setResource(Resource resource) {
      this.resource = resource;
   }
}
```

org.aspectj.lang.annotation.Before的value成员变量必须指定，可以像代码清单10-8那样直接指定Pointcut表达式，也可以指定单独声明的Pointcut定义的Pointcut Signature，如代码清单10-10所示。

代码清单10-10 使用Pointcut Signature引用Pointcut的Before Advice示例

```
@Aspect
public class ResourceSetupJAspect {
  private Resource resource;

  @Pointcut("execution(boolean *.execute())")
  private void resourceSetupJoinpoints(){}

  @Before("resourceSetupJoinpoints()")
  public void setupResourcesBefore() throws Throwable
  {
    if(!getResource().exists())
    {
      FileUtils.forceMkdir(getResource().getFile());
    }
  }
  ...
}
```

某些情况下，我们可能需要在Advice定义中访问Joinpoint处的方法参数，在1.*x*的Spring AOP中，我们可以通过MethodBeforeAdvice的before方法传入的Object[]数组，访问相应的方法参数。现在，我们有如下两种方式达到相同的目的。

- **通过org.aspectj.lang.JoinPoint。**在@AspectJ形式的Aspect中，定义Before Advice的方法可以将第一个参数声明为org.aspectj.lang.JoinPoint类型，通过Joinpoint我们可以借助它的getArgs()方法，访问相应Joinpoint处方法的参数值。另外，我们还可以借助它的其他方法取得更多信息，比如，getThis()获得当前代理对象，getTarget()取得当前目标对象，getSignature()取得当前Joinpoint处的方法签名等。

  这样，通过如下形式的方法定义，我们就可以借助org.aspectj.lang.JoinPoint取得需要的信息：

  ```
  public void setupResourcesBefore(JoinPoint joinpoint) throws Throwable{
    joinpoint.getArgs();
    //...
  }
  ```

  当然，通常情况下，如果不需要这些信息，org.aspectj.lang.JoinPoint的方法参数声明是可以省略的。

- **通过args标志符绑定。**前面已经介绍过了args标志符，我们可以通过为其指定相应的对象类型，来进一步限定Pointcut定义。不过，它还有一个用处：当args标志符接受的不是具体的对象类型，而是某个参数名称的时候，它会将这个参数名称对应的参数值绑定到对Advice方法的调用。

  我们可以在Before Advice对应的方法之上声明需要的参数（任何必要的类型），然后在Pointcut定义中使用args将这些声明的参数绑定到调用即可，如下：

  ```
  @Before(value="execution(boolean *.execute(String,..)) && args(taskName)")
  public void setupResourcesBefore(String taskName) throws Throwable
  ```

10

```
{
  // 访问'taskName'参数
  ...
}
```

注意，args指定的参数名称必须与Advice定义所在方法的参数名称相同。在这里，args指定的值和setupResourcesBefore方法的参数名都是taskName。

当然，如果Advice引用的是独立的Pointcut定义，使用args绑定的形式也是大同小异的。只不过现在Advice方法直接引用带有参数的Pointcut定义罢了，具体示例如下所示：

```
@Pointcut("execution(boolean *.execute(String,..)) && args(taskName)")
private void resourceSetupJoinpoints(String taskName){}

@Before(value="resourceSetupJoinpoints(taskName)")
public void setupResourcesBefore(String taskName) throws Throwable{
  ...
}
```

另外，我们还可以同时使用org.aspectj.lang.JoinPoint以及使用args参数名称绑定功能。这实际上是显而易见的，如下所示：

```
@Before(value="execution(boolean *.execute(String,..)) && args(taskName)")
public void setupResourcesBefore(JoinPoint joinpoint,String taskName) throws Throwable{
  // access 'taskName' argument
  ...
}
```

不过，需要注意，org.aspectj.lang.JoinPoint永远处于第一个参数位置。

针对Before Advice访问方法参数的两种方式，我们需要进一步扩展它们的应用场景。

- 首先，在@AspectJ形式的Advice声明方式中，不只Before Advice的方法第一个参数可以声明为org.aspectj.lang.JoinPoint类型，实际上，除了Around Advice和Introduction不可以这么用之外，余下的Advice类型的方法声明都遵循这个规则；
- 其次，不只可以使用args标志符来绑定参数声明到方法，实际上，Pointcut标志符中，除了execution标志符不会直接指定对象类型之外，其他像this、target、@within、@target、@annotation、@args等原本都是指定对象类型的。它们与args一样，在这样的场合下，如果它们指定的是参数名称，那么，所起的作用与args在这种参数绑定的场景中的作用是类似的，比如：

```
@Before(value="execution(boolean *.execute(String,..)) && args(taskName)&& ➥
@annotation(txInfo)")
public void setupResourcesBefore(String taskName,Transactional txInfo) throws Throwable{
  // 访问'taskName'参数
  // 通过txInfo参数访问事务控制相关信息
  ...
}
```

当目标对象上Joinpoint处的方法标注了org.springframework.transaction.annotation.Transactional，我们可以通过这种方式取得其事务设置的详细信息。

2. After Throwing Advice

After Throwing Advice的@AspectJ形式声明，同样是在Aspect定义类中声明相应的方法，只不过要使用org.aspectj.lang.annotation.AfterThrowing对其进行标注。

将ThrowsAdvice部分的实例ExceptionBarrierThrowsAdvice 移植到@AspectJ形式的声明，如

代码清单10-11所示。

代码清单10-11 @AspectJ形式的After Throwing Advice声明示例

```
@Aspect
public class ExceptionBarrierAspect {

  private JavaMailSender mailSender;
  private String[]   receiptions;

  @AfterThrowing(pointcut="execution(boolean *.execute(String,..))", ➥
  throwing="e")
  public void afterThrowing(RuntimeException e)
  {
    final String exceptionMessage = ExceptionUtils.getFullStackTrace(e);
    getMailSender().send(new MimeMessagePreparator(){
    public void prepare(MimeMessage message) throws Exception {
      MimeMessageHelper helper = new MimeMessageHelper(message);
      helper.setSubject("...");
      helper.setTo(getReceiptions());
      helper.setText(exceptionMessage);
      }});
  }

  public JavaMailSender getMailSender() {
    return mailSender;
  }

  public void setMailSender(JavaMailSender mailSender) {
    this.mailSender = mailSender;
  }

  public String[] getReceiptions() {
    return receiptions;
  }

  public void setReceiptions(String[] receiptions) {
    this.receiptions = receiptions;
  }
}
```

@AfterThrowing有一个独特的属性，即throwing，通过它，我们可以限定Advice定义方法的参数名，并在方法调用的时候，将相应的异常绑定到具体方法参数上。在这里，我们使用throwing="e"将RuntimeException类型的异常绑定到了afterThrowing方法，这样就可以访问具体的异常信息。

当然，根据不同的场景，我们可以对代码清单10-10的代码示例进行增删。如果不需要访问具体异常，那么就声明没有任何参数的方法。这样，也不用明确指定注解的哪个属性对应哪个值，如下所示：

```
@AfterThrowing("execution(boolean *.execute(String,..))")
public void afterThrowing(){
  ...
}
```

不过，如果不但要访问具体的异常信息，还需要访问其他信息，那么我们就可以像本节的第1小节中所说的那样，结合这两种方式来获取必要的信息，如下所示：

```
@AfterThrowing(pointcut="execution(boolean *.execute(String,..))", throwing="e")
public void afterThrowing(Joinpoint joinpoint,RuntimeException e)
```

10

```
{
   ...
}
```

是使用Joinpoint的一站式服务，还是使用args等标志符的专职服务，完全看个人的喜好了。

> **注意**　与1.x中的ThrowsAdvice可以同时针对不同的异常类型声明不同的方法进行处理一样，我们也可以在Aspect中针对多个不同的异常类型，声明不同的After Throwing Advice的方法定义。

3. After Returning Advice

要声明@AspectJ形式的After Returning Advice，使用org.aspectj.lang.annotation.AfterReturning标注Aspect定义中的相应方法即可。

如果我们将之前的TaskExecutionAfterReturningAdvice 移植过来，那么看起来如代码清单10-12所示。

代码清单10-12　@AspectJ形式的After Returning Advice声明示例

```
@Aspect
public class TaskExecutionCompletionAspect {
   private SqlMapClientTemplate sqlMapClientTemplate;

   @AfterReturning("execution(boolean *.execute(String,..))")
   public void taskExecutionCompleted(JoinPoint jp)
   {
      Class clazz = jp.getTarget().getClass();
      getSqlMapClientTemplate()
         .insert("BATCH.insertTaskStatus",clazz.getName());
   }

   public SqlMapClientTemplate getSqlMapClientTemplate() {
      return sqlMapClientTemplate;
   }

   public void setSqlMapClientTemplate(SqlMapClientTemplate sqlMapClientTemplate) {
      this.sqlMapClientTemplate = sqlMapClientTemplate;
   }
}
```

这里，我们只通过声明Joinpoint参数来获取目标对象的信息。但对于After Returning Advice来说，某些时候需要访问方法的返回值，这时，可以通过@AfterReturning的returning属性将返回值绑定到After Returning Advice定义所在的方法，如下所示：

```
@AfterReturning(pointcut="execution(boolean *.execute(String,..))",returning="retValue")
public void taskExecutionCompleted(JoinPoint jp,boolean retValue)
{
   // 通过'retValue'参数访问返回值
   ...
}
```

关于其他参数绑定的方式，可以参照本节第1小节中的内容，此处不做赘述。

4. After（Finally）Advice

最初1.x版本发布的Spring AOP中没有专门针对After(Finally) Advice类型的接口定义，所以，通常我们会通过使用Around Advice（或者说拦截器）来达到与使用After(Finally) Advice相同的目的。2.0之后，我们通过@AspectJ形式的AOP就能够在Aspect中直接声明这种类型的Advice。

要声明@AspectJ形式的After(Finally) Advice，只需要通过org.aspectj.lang.annotation.After对Aspect中定义的相应方法进行标注。

对于匹配的Joinpoint处的方法执行来说，不管该方法是正常执行返回，还是执行过程中抛出异常而非正常返回，都会触发其上的After(Finally) Advice执行。所以，After(Finally) Advice适合用于释放某些系统资源的场景。

假设我们的系统中，某个目录存放的都是临时文件，系统退出后或者某些合适的时机，这些临时文件需要删除以释放磁盘空间，那么，我们可以定义一个After(Finally) Advice来做类似的工作（见代码清单10-13）。

代码清单10-13 @AspectJ形式的After(Finally) Advice声明示例

```
@Aspect
public class SystemCleanerAspect {
  private Resource resource;

  @After("execution(boolean *.execute(String,..))")
  public void cleanUpResourcesIfNecessary() throws Throwable
  {
    if(getResource().exists())
    {
      FileUtils.forceDelete(getResource().getFile());
    }
  }

  public Resource getResource() {
    return resource;
  }

  public void setResource(Resource resource) {
    this.resource = resource;
  }
}
```

当然，更多的时候，我们可以使用After(Finally) Advice来处理网络连接的释放、数据库资源的释放等。

如果处理逻辑中需要访问Joinpont处的方法参数相关信息，可以通过Before Advice部分提及的两种方式进行处理。

5. Around Advice

@AspectJ形式的Around Advice声明与之前几个Advice类型的声明类似，最大的区别就是用于标注具体Advice定义方法的注解不一样，我们使用org.aspectj.lang.annotation.Around来标注Aspect中Around Advice所定义的方法。

不过，Around Advice的方法定义与之前几个Advice类型的方法定义有一些不同。我们说过，Before Advice、After Throwing Advice、After Returning Advice 以及After Advice的方法定义第一个参数可以为org.aspectj.lang.JoinPoint类型，而且是可选的。而对于Around Advice的方法定义来说，它的第一个参数必须是org.aspectj.lang.ProceedingJoinPoint类型（属于org.aspectj.lang.JoinPoint的子类），而且必须指定。通常情况下，我们需要通过ProceedingJoinPoint的proceed()方法继续调用链的执行。

将PerformanceMethodInterceptor 逻辑移植到@AspectJ形式的Around Advice声明形式，如代码清单10-14所示。

代码清单10-14 @AspectJ形式的Around Advice声明示例

```
@Aspect
public class PerformanceTraceAspect{
  private final Log logger = LogFactory.getLog(PerformanceTraceAspect.class);

  @Around("execution(boolean *.execute(String,..))")
  public Object performanceTrace(ProceedingJoinPoint joinpoint) throws Throwable
  {
    StopWatch watch = new StopWatch();
    try
    {
      watch.start();
      return joinpoint.proceed();
    }
    finally
    {
      watch.stop();
      if(logger.isInfoEnabled())
        logger.info(watch.toString());
    }
  }
}
```

当然，我们可以根据情况来决定是否要调用proceed()方法，甚至调用多少次。

除了直接调用proceed()方法，还可以在调用proceed()方法的时候，传入一个Object[]数组代表方法参数列表。通常，在修改方法调用参数值的时候，会使用带Object[]参数的proceed()方法。

同样以PerformanceTraceAspect为例，我们为Around Advice的方法定义增加参数，代表当前任务的名称。在继续执行PerformanceTraceAspect后面调用链上的Advice逻辑或者目标方法之前，我们会在PerformanceTraceAspect内先对传入的参数进行处理，然后再继续调用流程，详情如代码清单10-15所示。

代码清单10-15 带有Object[]方法参数的proceed方法使用示例

```
@Around("execution(boolean *.execute(String,..)) && args(taskName)")
public Object performanceTrace(ProceedingJoinPoint joinpoint,String taskName) throws Throwable
{
  StopWatch watch = new StopWatch();
  try
  {
    String modifiedTaskName = processTaskName(taskName);
    watch.start();
    return joinpoint.proceed(new Object[]{modifiedTaskName});
  }
  finally
  {
    watch.stop();
    if(logger.isInfoEnabled())
      logger.info(watch.toString());
  }
}
```

如果有多个参数，应该按照方法定义的顺序指定，以便定义的Around Advice可以在Spring AOP以及AspectJ之间通用。

6. Introduction

以@AspectJ形式声明的Introduction，完全不同于之前提过的任何Advice，这回不是使用注解对

Aspect中各Advice的定义方法进行标注，而是对Aspect中的实例变量定义进行标注。

我们知道，在Spring中，Introduction的实现是通过将需要添加的新的行为逻辑，以新的接口定义增加到目标对象上。要以@AspectJ形式声明Introduction，我们需要在Aspect中声明一个实例变量，它的类型对应的就是新增加的接口类型，然后使用org.aspectj.lang.annotation. DeclareParents对其进行标注。通过@DeclareParents指定新接口定义的实现类以及将要加诸其上的目标对象。

下面我们来看一个例子。

以讲解ProxyFactoryBean部分的ITask以及ICounter作为实例对象，将ICounter的行为逻辑加到ITask类型的目标实现类上。假设ITask的实现类是MockTask，而ICounter的实现类是CounterImpl，我们通过如下的Aspect声明将ICounter的行为introduce附加到ITask之上：

```
@Aspect
public class IntroductionAspect {

   @DeclareParents(
        value="...MockTask",
        defaultImpl=CounterImpl.class
   )
   public ICounter counter;

}
```

该定义需要注意如下几个地方。

- @DeclareParents所归属的域定义类型为ICounter，也就是将为目标对象新增加的对象类型。
- 通过@DeclareParents的value属性，可以指定将要应用到的目标对象。现在只指定了单一的目标对象，还可以通过通配符指定一批目标对象。假设cn.spring21.unveilspring. service包下的所有类都想增加ICounter的行为，那么，可以参照如下代码为value指定表达式：

```
@DeclareParents(
     value="cn.spring21.unveilspring.service.*",
     defaultImpl=CounterImpl.class
)
public ICounter counter;
```

- 通过@DeclareParents的defaultImpl属性，我们可以指定新增加的接口定义的实现类。在这里，ICounter的实现类为CounterImpl，所以可以指定defaultImpl=CounterImpl.class。

如果将这个Aspect定义按照如下代码注册到Spring的IoC容器：

```
<aop:aspectj-autoproxy proxy-target-class="true">
</aop:aspectj-autoproxy>

<bean id="task" class="...MockTask" scope="prototype">
</bean>

<bean id="counterInroduction" class="...IntroductionAspect"></bean>
```

并且通过编程形式对目标对象进行访问，如以下代码所示：

```
ApplicationContext ctx = new ClassPathXmlApplicationContext("...");
Object task1 = ctx.getBean("task");
Object task2 = ctx.getBean("task");

System.out.println(((ICounter)task1).getCounter());
```

```
System.out.println(((ICounter)task1).getCounter());
System.out.println(((ICounter)task2).getCounter());
```

会发现，不同的目标对象现在对应着不同的ICounter实例，因为每个方法调用后返回的状态都是各自的。

> **注意**　Introduction属于per-instance类型的Advice，所以，还是不要忘记，目标对象的scope通常情况下应该设置为prototype。当然啦，某些情况下，可能会特意设置为singleton的scope，也未可知。

10.1.4　@AspectJ 中的 Aspect 更多话题

我想关于@AspectJ形式的Aspect声明就没有必要再重复一遍了，简单的POJO+@Aspect就是一个@AspectJ形式的Aspect。不过，跟Aspect有关的几个话题倒是有必要进一步讲述一下。

1. Advice的执行顺序

对于多个Advice来说，如果它们引用的Pointcut定义恰好匹配同一个Joinpoint的时候，在这同一个Joinpoint上，这些Advice该按照什么顺序执行呢？对于这个问题，我们应该从以下两个角度来看。

- 当这些Advice都声明在同一个Aspect内的时候。如果匹配同一个Joinpoint的多个Advice都声明在同一个Aspect定义中，那么这些Advice的执行顺序，由它们在Aspect中的声明顺序决定。最先声明的Advice拥有最高的优先级。对于Before Advice来说，拥有最高优先级的最先运行，而对于AfterReturngingAdvice，拥有最高优先级的则是最后运行。

  假设声明拥有多个Advice定义的Aspect（如代码清单10-16所示），这些Advice全部引用同一个Pointcut定义。

代码清单10-16　位于同一个Aspect内并且匹配同一个Joinpoint的Advice声明示例

```
@Aspect
public class MultiAdvicesAspect {

   @Pointcut("execution(boolean *.execute(String,..))")
   public void taskExecution(){}

   @Before("taskExecution()")
   public void beforeOne()
   {
      System.out.println("before One");
   }

   @Before("taskExecution()")
   public void beforeTwo()
   {
      System.out.println("before Two");
   }

   @AfterReturning("taskExecution()")
   public void afterReturningOne()
   {
      System.out.println("after returning one");
   }

   @AfterReturning("taskExecution()")
```

```
    public void afterReturningTwo()
    {
      System.out.println("after returning two");
    }
}
```

如果将该Aspect织入目标对象，那么可以得到如下的输出结果：

```
before One
before Two
task executed.
after returning two
after returning one
```

Before Advice的输出很容易理解，尤其需要注意After Returning Advice的输出，afterReturningOne()先于afterReturningTwo()声明，拥有较高的优先级，但因为它属于After Returning Advice，所以，要晚于afterReturningTwo()的执行。其他类型的Advice执行可以按照这个规则依此类推。

- **当这些Advice声明在不同的Aspect内的时候。**如果多个Advice声明所对应的Pointcut定义匹配同一个Joinpoint，但它们又不是声明在同一个Aspect内的话，情况是怎么样的呢？这时，我们需要用到Spring的org.springframework.core.Ordered接口，只需要让相应的Aspect定义实现Ordered接口即可，否则，Advice的执行顺序是不确定的。

Ordered的语义与9.4.3节所讲述的一样，Ordered.getOrder()方法返回较小值的Aspect，其内部所声明的Advice拥有较高优先级，而Ordered.getOrder()方法返回值较大的Aspect，其内部声明的Advice则拥有较低的执行优先级。

如果再声明一个Aspect（见代码清单10-17），让它的一个Before Advice也引用MultiAdvicesAspect定义的Pointcut。

代码清单10-17 AnotherAspect定义

```
public class AnotherAspect implements Ordered{
  private final Log logger = LogFactory.getLog(PerformanceTraceAspect.class);

  @Before("package.MultiAdvicesAspect.taskExecution()")
  public void doBefore()
  {
    System.out.println("before in AnotherAspect");
  }

  ...

  public int getOrder() {
    return 100;
  }
}
```

同时，让MultiAdvicesAspect也实现Ordered接口，并先将getOrder()的返回值定为20，那么当将这两个Aspect同时织入同一个目标对象的时候，输出就会变为：

```
before One
before Two
before in AnotherAspect
task executed.
after returning two
after returning one
```

因为MultiAdvicesAspect比AnotherAspect拥有的执行优先级较高，所以它内部的两个Before Advice要比AnotherAspect内部的Before Advice优先执行。

小心　当使用org.springframework.core.Ordered对Aspect内的Advice执行顺序进行限定时，如下两种情况需要注意。

- 如果使用Spring的IoC容器注册并使用这些Aspect，让自动代理机制处理这些横切逻辑到目标对象的织入，那么，多个Aspect内的Advice执行顺序正是如以上所声明的那样。
- 如果通过编程的方式来使用这些Aspect，那么，情况就大不一样了，Aspect内的Advice执行顺序完全由添加到AspectJProxyFactory的顺序来决定，而不是Ordered接口所规定的顺序。如果采用如下代码将MultiAdvicesAspect和AnotherAspect织入目标对象的话，AnotherAspect内的Advice要比MultiAdvicesAspect内的Advice优先执行:

```
AspectJProxyFactory weaver = new AspectJProxyFactory();
weaver.setProxyTargetClass(true);
weaver.setTarget(new MockTask());
weaver.addAspect(new AnotherAspect());
weaver.addAspect(new MultiAdvicesAspect());
...
MockTask task = (MockTask)weaver.getProxy();
task.execute(null);
```

如果先通过addAspect添加MultiAdvicesAspect，后添加AnotherAspect，才会跟使用Ordered接口并通过自动织入所达到的效果一样。

---

2. Aspect的实例化模式

对于注册到容器的各个Aspect，它们默认的实例化模式（singleton instantiation model）采用的是singleton，也就是说，在容器中会实例化并持有每个Aspect定义的单一实例（行为上恰好与容器的singleton型scope的行为相吻合）。

除了singleton的实例化模式，AspectJ还支持perthis、pertarget、percflow、percflowbelow以及perwithin等实例化模式。不过，Spring 2.0之后的AOP只支持默认的singleton、perthis和pertarget三种实例化模式。

要想指定相应Aspect的实例化模式，可以通过@Aspect指定perthis或者pertarget语句。如果想让MultiAdvicesAspect的实例化模式从默认的singleton变为perthis，可以如下指定：

```
@Aspect("perthis(execution(boolean *.execute(String,..)))")
public class MultiAdvicesAspect {

  @Pointcut("execution(boolean *.execute(String,..))")
  public void taskExecution(){}

  ...
}
```

这样，在perthis指定的Pointcut定义匹配之后，会为相应的代理对象实例化各自的Aspect实例。对于pertarget来说，则是为匹配的单独的目标对象实例化相应Aspect实例。（可以在perthis或者pertarget语句中，使用Pointcut定义的Pointcut Signature代替直接的Pointcut表达式定义。）

不过，使用perthis或者pertarget指定了Aspect的实例化模式之后，将这些Aspect注册到容器时，不能为其bean定义指定singleton的scope，否则会出现异常。毕竟，容器先限定了只有一个实例，就不能为每一个代理对象或者目标对象实例化相应实例。

```
<aop:aspectj-autoproxy proxy-target-class="true">
</aop:aspectj-autoproxy>

<bean id="task" class="...MockTask" scope="prototype">
</bean>


<bean id="multiAdvicesAspect" class="...MultiAdvicesAspect" scope="prototype"></bean>
```

当然，如以上代码所示，为了perthis和pertarget能够正常工作，我们也应该考虑将相应的目标对象的scope设置为prototype或者其他自定义scope类型。

10.2 基于 Schema 的 AOP

基于Schema的AOP是Spring 2.0发布之后新增加的一种AOP使用方式，我们可以从如下两个角度来看待基于Schema的AOP。

- **配置方式的改变**。Spring框架从1.*x*版本升级到2.*x*版本之后，提倡的容器配置方式从基于DTD的XML转向了基于Schema的XML，进一步提高了配置方式的灵活性和可扩展性。同时，新的基于Schema的配置方式为Spring的AOP功能专门提供了独有的命名空间。原来1.*x*中基于DTD的AOP配置方式，可以稍微转换一下配置方式就移植到基于Schema的AOP，所以，从这一点来说，基于Schema的AOP只是配置方式的改变。当然，这只是从本质上说，稍后我们将看到，还有少许新的AOP特性实际上是可用的。
- **@AspectJ形式AOP的折中**。要使用@AspectJ形式的AOP，必须要求使用Java 5或者更高版本的JDK或JRE，因为注解是Java 5发布之后才引入的特性。如果我们不得不使用Java 5之前的版本，而又想使用基于POJO的Aspect声明方式，这时，就可以使用基于Schema的Spring AOP。使用基于Schema的AOP，我们可以依然使用POJO声明Aspect以及相关的Advice。不过，不需要注解标注了，直接通过Shema的配置文件进行配置就可以，@AspectJ形式的Pointcut表达式也全都可以配置到基于Schema的配置文件中。

要使用基于Schema的AOP，IoC容器的配置文件应该使用基于Schema的XML，同时在文件头中增加针对AOP的命名空间声明，如代码清单10-18所示。

代码清单10-18 添加使用aop命名空间的配置文件示例

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" ➥
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ➥
xmlns:aop="http://www.springframework.org/schema/aop" ➥
xsi:schemaLocation="http://www.springframework.org/schema/beans ➥
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd ➥
http://www.springframework.org/schema/aop ➥
http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

   <!--相关bean定义-->

</bean>
```

添加了aop的命名空间声明之后，我们就可以使用aop命名空间内的各种配置元素了。

10.2.1 基于 Schema 的 AOP 配置概览

新的基于Schema的AOP配置方式，针对Pointcut、Advisor以及Aspect等概念提供了独立的配置元

10

素，所有这些配置元素都包含在统一的配置元素中，即<aop:config>，如代码清单10-19所示。

代码清单10-19　<aop:config>配置示例

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" ➥
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ➥
xmlns:aop="http://www.springframework.org/schema/aop" ➥
xsi:schemaLocation="http://www.springframework.org/schema/beans ➥
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd ➥
http://www.springframework.org/schema/aop ➥
http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

  <aop:config>
    <!--其他aop概念元素定义-->
  </aop:config>

</bean>
```

<aop:config>只有一个属性proxy-target-class，对应ProxyConfig中的proxyTargetClass属性，通过该属性，我们可以控制是使用基于接口的代理还是基于类的代理，如代码清单10-20所示。

代码清单10-20　<aop:config>的proxy-target-class属性配置示例

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" ➥
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ➥
xmlns:aop="http://www.springframework.org/schema/aop" ➥
xsi:schemaLocation="http://www.springframework.org/schema/beans ➥
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd ➥
http://www.springframework.org/schema/aop ➥
http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

  <aop:config proxy-target-class="false">
    <!--其他aop概念元素定义-->
  </aop:config>
</bean>
```

<aop:config>内部可以有三个子元素，分别是<aop:pointcut、<aop:advisor>和<aop:aspect>。它们三个必须按照顺序进行配置，如代码清单10-21所示。

代码清单10-21　<aop:config>内部子元素配置示例

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" ➥
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ➥
xmlns:aop="http://www.springframework.org/schema/aop" ➥
xsi:schemaLocation="http://www.springframework.org/schema/beans ➥
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd ➥
http://www.springframework.org/schema/aop ➥
http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

  <aop:config proxy-target-class="false">
    <aop:pointcut/>
    <aop:advisor/>
    <aop:aspect></aop:aspect>
  </aop:config>

</bean>
```

在这个顺序的基础上，我们可以根据需要声明多个并列的相同元素。

另外，我们还可以在同一个配置文件中配置多个<aop:config>，如代码清单10-22所示。

代码清单10-22 使用多个<aop:config>的配置演示

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" ➥
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ➥
xmlns:aop="http://www.springframework.org/schema/aop" ➥
xsi:schemaLocation="http://www.springframework.org/schema/beans ➥
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd ➥
http://www.springframework.org/schema/aop ➥
http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

  <aop:config proxy-target-class="false">
    <aop:pointcut/>
    <aop:advisor/>
    <aop:aspect></aop:aspect>
  </aop:config>

  <aop:config proxy-target-class="true">
    <aop:pointcut/>
    <aop:advisor/>
    <aop:aspect></aop:aspect>
  </aop:config>

  <!-- ... -->

</bean>
```

对于<aop:config>来说，底层基本上是使用1.*x*中的自动代理机制实现的。相应的自动代理实现类，会根据该元素内部对应的Pointcut、Advisor以及Aspect的子元素取得必要的织入信息，然后为容器内注册的bean进行自动代理。所以，如果愿意，不用<aop:config>，而依然使用明确指定AutoProxyCreator实现类的方式也是可以的。

> **警告** 通常情况下，使用<aop:config>和直接使用AutoProxyCreator两种方式最好不要一起使用，因为这可能造成一些问题，比如某些Advice逻辑没有织入系统。在Spring的参考文档和Support论坛上都提到过这个问题。

在大体了解了基于Schema的配置方式之后，让我们来深入看一下各个元素的配置与具体开发之间的关系吧！

10.2.2 向基于 Schema 的 AOP 迁移

1. 单纯的迁移

1.*x*版本的Spring AOP通过Advisor的概念对横切关注点进行封装。当把相应的Pointcut定义和Advice定义注册到容器之后（通常是在基于DTD的XML配置文件中），通过声明相应的Advisor实现，将这些Pointcut以及Advice定义装配到一起，最后通过某个AutoProxyCreator进行最后的织入。

在转向2.*x*版本基于Schema的配置方式之后，这些概念实际上是相同的，唯一需要改变的是具体配置方式的改变。现在，使用<aop:advisor>代替各种具体的Advisor实现类的bean定义声明，使用<aop:config>取代各种AutoProxyCreator。

如果将PerformanceMethodInterceptor通过基于Schema的方式进行配置，那么配置文件看起来将如代码清单10-23所示。

代码清单10-23　基于Schema的AOP配置示例

```
<aop:config>
  <aop:advisor id="performanceAdvisor" ➥
  pointcut-ref="targetPointcut" ➥
  advice-ref="performanceInterceptor" ➥
  order="1"/>
</aop:config>

<bean id="targetPointcut" class="org.springframework.aop.support.JdkRegexpMethodPointcut">
  <property name="pattern" value="..."/>
</bean>

<bean id="performanceInterceptor" class="...PerformanceMethodInterceptor">
</bean>

...
```

现在的配置看起来是不是要更具描述性呢？

- 对于PerformanceMethodInterceptor和它对应的Pointcut的bean定义来说，与之前基于DTD的配置方式没有什么改变，按照通常的注册方式就可以。当然，对于Pointcut的定义来说，我们可以替换任何我们喜欢用的Pointcut实现类。
- 在<aop:config>中使用<aop:advisor>配置相应的Advisor，也就是特定于Spring AOP的Aspect。<aop:advisor>的如下几个属性可供使用。
  - id。指定当前Advisor定义的标志id。
  - pointcut-ref。通过这个属性指定当前Advisor所对应的Pointcut定义是什么，需要指定容器中注册的具体的Pointcut对象引用。
  - advice-ref。指定当前Advisor对应的Advice对象引用。
  - order。指定当前Adviosr的顺序号，因为基本上所有的Advisor实现都实现了Ordered接口。

基本上，<aop:advisor>就是一个个具体的Advisor的bean定义的对等体。

到目前为止，<aop:advisor>和<aop:config>可以不加任何条件地满足我们之前的AOP配置需求。不过，它们的组合可不仅限于这些。

2. 深入挖掘<aop:advisor>

对于<aop:advisor>所使用的Pointcut定义来说，除了可以使用pointcut-ref来指定具体的1.x版本中各种Pointcut实现类的引用，我们还有如下两种选择。

- *使用pointcut而不是pointcut-ref*

<aop:advisor>还有一个pointcut属性，但它不对之前的aop开放。通过该属性，我们只能指定AspectJ形式的Pointcut表达式，所以，如果要使用它，不要忘记将AspectJ的类库添加到应用程序的ClassPath中。代码清单10-24演示了<aop:advisor>的pointcut属性的使用。

代码清单10-24　<aop:advisor>的pointcut属性的使用示例

```
<aop:config>
  <aop:advisor id="performanceAdvisor" ➥
  pointcut="execution(public void *.doSomething(..))" ➥
  advice-ref="performanceInterceptor" ➥
  order="1"/>
```

```
</aop:config>

<bean id="performanceInterceptor" class="...PerformanceMethodInterceptor">
</bean>

...
```

现在，直接通过pointcut这个属性指定AspectJ形式的Pointcut表达式就可以，不用为了当前的Advisor定义单独声明一个Pointcut实现类的bean定义引用。

● 使用<aop:pointcut>

前面讲述过，<aop:config>内部可以有一个或者多个<aop:pointcut>声明，这些声明因为是独立定义的，所以，可以被<aop:advisor>甚至<aop:aspect>所引用。

可以依然使用<aop:advisor>的pointcut-ref属性指定Pointcut定义的引用。不过，这次，不是指向1.x版本中独立声明的Pointcut对象引用，而是指向<aop:pointcut>，如代码清单10-25所示。

代码清单10-25 结合<aop:pointcut>的<aop:advisor>的使用示例

```
<aop:config>
  <aop:pointcut id="myPointcut" ➥
  type="regex" ➥
  expression="cn\.spring21\..*\.doSomething"/>

  <aop:advisor id="performanceAdvisor"
       pointcut-ref="myPointcut"
       advice-ref="performanceInterceptor"
       order="1"/>
</aop:config>

<bean id="performanceInterceptor" class="...PerformanceMethodInterceptor">
</bean>

...
```

如果指定<aop:pointcut>的type为regex，那么可以为expression指定正则表达式形式的Pointcut描述。如果指定<aop:pointcut>的type为aspectj，那么可以为expression指定AspectJ形式的Pointcut表达式。默认情况下，不明确指定type，将按照AspectJ形式的Pointcut表达式进行解析。不过，不得不提到的是，关于type的语义只是理论上的，实际上，当指定type的值为regex时，框架在处理表达式解析的时候依然采用的是AspectJ形式的解析规则。我在Spring Support论坛上提到过这个问题，回答是AopNamespaceHandler 可能忽略了这种情况的处理。所以，如果这个问题在新的Spring版本中没有修正，应该避免使用type为regex的形式（JIRA SPR-4180 issue）。

> **注意** 通过type="regex"的<aop:pointcut> + <aop:advisor>，或者<aop:advisor> + 独立的Pointcut对象声明，是在以新的配置方式来使用1.x版本的AOP，没有任何附加；通过type="aspectj"的<aop:pointcut> + <aop:advisor>，或者直接通过pointcut属性来使用<aop:advisor>，则是扩展了1.x版本的AOP，可以让1.x版本的AOP使用AspectJ形式的Pointcut表达式。

10.2.3 @AspectJ 到“基于 Schema 的 AOP”迁移

如果你想使用@AspectJ形式的Aspect和Advice声明方式，而又不能使用Java 5甚至更高版本的

JVM，或者你就是想使用基于XML配置文件的AOP使用方式，那么恭喜你，基于Schema的AOP可以满足这方面的要求。

@AspectJ形式的AOP采用相应的注解来标注必要的信息，如哪个对象是Aspect定义，Aspect定义中哪个方法对应的是什么类型的Advice声明，或者Pointcut的定义是什么等。在我们使用基于Schema的AOP之后，Aspect和Advice的定义还是基于POJO的，不过那些通过注解表达的信息全部都移到了XSD形式的XML配置文件中。

让我们详细看一下原来@AspectJ形式的各种实体，在基于Schema的AOP中是如何表达的。

1. 基于Schema的Aspect声明

基于Schema的Aspect声明由两部分组成：Aspect的定义和Aspect到容器的配置。

- Aspect的定义

我们依然使用POJO来声明Aspect，只不过，用不用@Aspect来标注这个POJO已经不再重要。因为Aspect的信息将通过Shema配置文件来捕捉。

使用基于Schema的AOP之后，我们的Aspect定义就是一个最为普通的POJO了，如以下代码所示：

```
public class SchemaBasedAspect{
   public void method1(){...}
   public void method2(){...}
   ...
}
```

- Aspect定义到容器的配置

将SchemaBasedAspect像通常的bean定义那样注册到容器之后，我们需要使用<aop:aspect>来引用它，如下所示：

```
<aop:config>
   <aop:aspect id="myAspect" ref="shemaBasedAspect" order="2">
      ...
   </aop:aspect>
</aop:config>

<bean id="shemaBasedAspect" class="...SchemaBasedAspect"></bean>
```

<aop:aspect>是<aop:config>的子元素，这在前面讲述过，它有三个属性。

- id。Aspect定义在配置文件中的标志id。
- ref。通过ref指向Aspect定义对应的容器内的bean定义。
- order。Aspect定义对应的顺序号，用于控制Aspect内定义的Advice在同一个Joinpoint的执行顺序。

2. 基于Schema的Pointcut声明

基于Schema的Pointcut声明（使用<aop:pointcut>）可以位于两个位置：一个位置是直接声明到<aop:config>下面，这样的Pointcut定义可以在其余的Advisor定义和Aspect定义中共享引用，另一个位置就是<aop:aspect>元素内部，这种Pointcut只能在其所声明的<aop:aspect>内部引用，相当于修饰符为private的定义。不管<aop:pointcut>声明位于什么位置，它们的可配置项是相同的，唯一不同的就是它们可以引用的范围。

关于<aop:config>下的<aop:pointcut>，我们在10.2.2节的第2小节已经讲述过了，而在<aop:aspect>下声明<aop:pointcut>与之没有任何不同，如下所示：

```
<aop:config>
```

```
    <aop:aspect id="myAspect" ref="shemaBasedAspect" order="2">
      <aop:pointcut id="privatePointcut" ➥
      expression="execution(public void *.doSth())"/>
      ...
    </aop:aspect>
  </aop:config>

  <bean id="shemaBasedAspect" class="...SchemaBasedAspect"></bean>
```

通常情况下，直接通过expression指定Pointcut表达式就可以了。如果我们使用了Java 5或者更高版本，而且以@AspectJ形式声明了Aspect，那么还可以在expression中指定Pointcut定义的Pointcut Signature，如下所示：

```
<aop:pointcut id="privatePointcut" ➥
expression="cn.spring21.unveilspring.SystemCommonsAspect.servicePointcut()"/>
```

以上内容假设SystemCommonsAspect类中以@AspectJ形式定义了系统中公用的Pointcut定义。

另外，因为通常Pointcut可以进行逻辑运算，AspectJ语法中的逻辑运算符是&&、||和!，而在XML中，&&是限制字符，所以，在基于Schema的AOP中，我们使用and、or和not来代替原始的逻辑运算符。当框架内部解析这些Pointcut表达式时，将会对表达式中的and、or和not关键字进行替换。

更多有关“基于Schama的AOP”中Pointcut声明所需要注意到的问题，可以参照最新的Spring参考文档或者AspecJ的开发文档。

3. 基于Schema的Advice声明

在基于Schema的AOP中的Advice声明也分为两部分，即Advice的定义和Advice的到容器的配置。

对于Advice的定义来说很简单，实际上就是Aspect定义类中的一个个方法定义，方法参数按照需要进行声明。除了Around Advice对应的方法定义第一个参数是ProceedingJoinPoint类型外，其他类型的Advice定义对应的方法声明没有太多说头。所以，我们重点来看一下这几种Advice类型在基于Schema的AOP中是如何配置的。

- Before Advice

基于Schema的AOP中Before Advice对应的配置元素是位于<aop:aspect>内的<aop:before>。假设我们在SchemaBasedAspect中声明的Before Advice定义如下所示：

```
public class SchemaBasedAspect {
  private final Log logger = LogFactory.getLog(SchemaBasedAspect.class);

  public void doBefore(JoinPoint jp)
  {
    if(logger.isInfoEnabled())
      logger.info("before method["+jp.getSignature().getName()+"] execution.");
  }
}
```

那么，可以通过<aop:before>通知AOP框架，SchemaBasedAspect中的doBefore方法就是我们声明的Before Advice定义，如下所示：

```
<aop:config>
  <aop:aspect id="myAspect" ref="shemaBasedAspect" order="2">
    <aop:pointcut id="privatePointcut" ➥
    expression="execution(public void *.doSth())"/>
    <aop:before pointcut-ref="privatePointcut" method="doBefore"/>
  </aop:aspect>
</aop:config>
```

```
<bean id="shemaBasedAspect" class="...SchemaBasedAspect"></bean>
```

我们通过<aop:before>的pointcut-ref指定Before Advice所引用的Pointcut定义。不过同样可以使用它的pointcut属性直接指定具体的Pointcut表达式。因为Advice定义属于Aspect内定义的方法，所以，配置Advice的时候可以通过<aop:before>的method属性指定具体的Before Advice对应的方法名称。

- After Returning Advice

如果我们在SchemaBasedAspect中声明一个对应After Returning Advice的方法定义，如代码清单10-26所示。

代码清单10-26 添加了AfterReturningAdvice的方法定义的SchemaBasedAspect

```
public class SchemaBasedAspect {
  private final Log logger = LogFactory.getLog(SchemaBasedAspect.class);

  public void doBefore(JoinPoint jp)
  {
    if(logger.isInfoEnabled())
      logger.info("before method["+jp.getSignature().getName()+ ➥
      "] execution.");
  }

  public void doAfterReturning(JoinPoint jp)
  {
    if(logger.isInfoEnabled())
      logger.info("method["+jp.getSignature().getName()+ ➥
      "] completed successfully.");
  }
  ...
}
```

那么，使用<aop:aspect>内的<aop:after-returning>对其进行配置如下：

```
<aop:config>
  <aop:aspect id="myAspect" ref="shemaBasedAspect" order="2">
    <aop:pointcut id="privatePointcut" ➥
    expression="execution(public void *.doSth())"/>
    <aop:before pointcut-ref="privatePointcut" method="doBefore"/>
    <aop:after-returning pointcut-ref="privatePointcut" method="doAfterReturning"/>
  </aop:aspect>
</aop:config>

<bean id="shemaBasedAspect" class="...SchemaBasedAspect"></bean>
```

当然，我们可能需要在After Returning Advice中访问方法的返回值，这时，可以通过<aop:after-returning>的returning属性指定返回参数名称，并在After Returning Advice对应的方法定义的参数列表中增加返回值的声明，如下代码演示了针对这种情况的处理：

```
<aop:after-returning pointcut-ref="privatePointcut" method="doAfterReturning" ➥
returning= "retValue"/>

public void doAfterReturning(JoinPoint jp,Object retValue)
{
  if(logger.isInfoEnabled())
  {
    logger.info("method["+jp.getSignature().getName()+"] completed successfully.");
    logger.info("with return value:"+retValue);
```

```
    }
}
```

唯一需要注意的就是通过returning指定的值，需要与方法声明中的参数名称相同。

- After Throwing Advice

After Throwing Advice的Schema声明使用<aop:aspect>内的<aop:after-throwing>元素。

如果我们在SchemaBasedAspect中声明一个用于处理RuntimeException的After Throwing Advice，如代码清单10-27所示。

代码清单10-27 添加了After Throwing Advice方法定义的SchemaBasedAspect

```
public class SchemaBasedAspect {
  private final Log logger = LogFactory.getLog(SchemaBasedAspect.class);

  public void doBefore(JoinPoint jp)
  {
    if(logger.isInfoEnabled())
      logger.info("before method["+jp.getSignature().getName()+ ➥
      "] execution.");
  }

  public void doAfterReturning(JoinPoint jp)
  {
    if(logger.isInfoEnabled())
      logger.info("method["+jp.getSignature().getName()+➥
      "] completed successfully.");
  }

  public void doAfterThrowing(RuntimeException e)
  {
    logger.error(ExceptionUtils.getFullStackTrace(e));
  }
  ...
}
```

对应它的Schema配置看起来如代码清单10-28所示。

代码清单10-28 <aop:after-throwing>相关配置示例

```
<aop:config>
  <aop:aspect id="myAspect" ref="shemaBasedAspect" order="2">
    <aop:pointcut id="privatePointcut" ➥
    expression="execution(public void *.doSth())"/>
    <aop:before pointcut-ref="privatePointcut" method="doBefore"/>
    <aop:after-returning pointcut-ref="privatePointcut" ➥
    method="doAfterReturning"/>
    <aop:after-throwing pointcut-ref="privatePointcut" ➥
    method="doAfterThrowing" throwing="e"/>
  </aop:aspect>
</aop:config>

<bean id="shemaBasedAspect" class="...SchemaBasedAspect"></bean>
```

同样需要注意的是，throwing指定的值要与方法定义的参数名称相同。

- After (Finally) Advice

After Advice同样用于释放系统资源相关场景，在基于Schema的AOP中，使用<aop:aspect>中的<aop:after>对其进行配置。

10

如果我们同样在SchemaBasedAspect中声明一个After Advice对应的方法定义（见代码清单10-29）。

代码清单10-29　添加了AfterAdvice定义的SchemaBasedAspect声明

```
public class SchemaBasedAspect {
  private final Log logger = LogFactory.getLog(SchemaBasedAspect.class);

  public void doBefore(JoinPoint jp)
  {
    if(logger.isInfoEnabled())
      logger.info("before method["+jp.getSignature().getName()+ ➥
      "] execution.");
  }

  public void doAfterReturning(JoinPoint jp)
  {
    if(logger.isInfoEnabled())
      logger.info("method["+jp.getSignature().getName()+➥
      "] completed successfully.");
  }

  public void doAfterThrowing(RuntimeException e)
  {
    logger.error(ExceptionUtils.getFullStackTrace(e));
  }

  public void doAfter()
  {
    logger.warn("release system resources ,etc.");
  }
  ...
}
```

那么增加<aop:after>配置后的文件内容如代码清单10-30所示。

代码清单10-30　<aop:after>相关配置示例

```
<aop:config>
  <aop:aspect id="myAspect" ref="shemaBasedAspect" order="2">
    <aop:pointcut id="privatePointcut" ➥
    expression="execution(public void *.doSth())"/>
    <aop:before pointcut-ref="privatePointcut" method="doBefore"/>
    <aop:after-returning pointcut-ref="privatePointcut" ➥
    method="doAfterReturning"/>
    <aop:after-throwing pointcut-ref="privatePointcut" ➥
    method="doAfterThrowing" throwing="e"/>
    <aop:after pointcut-ref="privatePointcut" method="doAfter"/>
  </aop:aspect>
</aop:config>

<bean id="shemaBasedAspect" class="...SchemaBasedAspect"></bean>
```

- **Around Advice**

我想看到这里你应该头都大了吧？除了配置使用的具体元素不同，实现和配置也都差不多嘛。对，不过，坚持一下，还剩一个。

Around Advice与之前的几个Advice类型的声明和配置都差不多，唯一的区别可能就是Around Advice的定义，即第一个参数必须是org.aspectj.lang.ProceedingJoinPoint类型，而且是必须指

定的，不像其他几个Advice对应的方法定义，第一个参数类型为org.aspectj.lang. JoinPoint，而且可以根据需要决定是否声明。

如果在SchemaBasedAspect中声明一个Around Advice定义如代码清单10-31。

代码清单10-31 添加了Around Advice定义的SchemaBasedAspect声明

```
public class SchemaBasedAspect {
  private final Log logger = LogFactory.getLog(SchemaBasedAspect.class);

  public void doBefore(JoinPoint jp)
  {
    if(logger.isInfoEnabled())
    logger.info("before method["+jp.getSignature().getName()+ ➥
    "] execution.");
  }

  public void doAfterReturning(JoinPoint jp)
  {
    if(logger.isInfoEnabled())
    logger.info("method["+jp.getSignature().getName()+ ➥
    "] completed successfully.");
  }

  public void doAfterThrowing(RuntimeException e)
  {
    logger.error(ExceptionUtils.getFullStackTrace(e));
  }

  public void doAfter()
  {
    logger.warn("release system resources ,etc.");
  }

  public Object doProfile(ProceedingJoinPoint pjp) throws Throwable
  {
    StopWatch watch = new StopWatch();
    try
    {
      watch.start();
      return pjp.proceed();
    }
    finally
    {
      watch.stop();
      if(logger.isInfoEnabled())
      {
        logger.info(watch);
      }
    }
  }
  ...
}
```

只需要通过<aop:aspect>中的<aop:around>对其进行配置就可以了（见代码清单10-32）。

代码清单10-32 <aop:around>相关配置示例

```
<aop:config>
  <aop:aspect id="myAspect" ref="shemaBasedAspect" order="2">
```

```
    <aop:pointcut id="privatePointcut" ➥
    expression="execution(public void *.doSth())"/>
    <aop:before pointcut-ref="privatePointcut" method="doBefore"/>
    <aop:after-returning pointcut-ref="privatePointcut" ➥
    method="doAfterReturning"/>
    <aop:after-throwing pointcut-ref="privatePointcut" ➥
    method="doAfterThrowing" throwing="e"/>
    <aop:after pointcut-ref="privatePointcut" method="doAfter"/>
    <aop:around pointcut-ref="privatePointcut" method="doProfile"/>
  </aop:aspect>
</aop:config>

<bean id="shemaBasedAspect" class="...SchemaBasedAspect"></bean>
```

注意 实际上，Around Advice对应的方法定义与@AspectJ形式的一样，唯一的不同就是后者使用@Around标注相关信息，而前者将这些信息移到了XML配置文件中。更多的信息可以参照10.1.3节的第5小节。

● Introduction

在原来@AspectJ形式的AOP中，我们通过在Aspect中为目标对象声明新增加的接口类型的实例变量，然后在这个变量基础上标注注解的方式来声明Introduction。现在转向基于Schema的AOP之后，我们不用在Aspect中定义任何东西，现在所有的信息全部通过Schema形式的配置文件进行配置。我们唯一要做的，就是根据Introdution的需要，实现新的接口以及该接口的实现类，然后通过基于Schema的配置文件将其注册到容器就可以了。

如果要为目标对象增加计数功能，那么我们在定义了ICounter接口和它的实现类CounterImpl之后，可以通过XSD形式的配置文件对Introduction的定义进行配置，如代码清单10-33所示。

代码清单10-33 基于Schema的Introduction相关配置示例

```
<aop:aspect id="myAspect" ref="genericShemaBasedAspect" order="2">
  <aop:pointcut id="privatePointcut" ➥
  expression="execution(public void *.doSth())"/>
  <aop:before pointcut-ref="privatePointcut" method="doBefore"/>
  <aop:after-returning pointcut-ref="privatePointcut" ➥
  method="doAfterReturning" returning="retValue"/>
  <aop:after-throwing pointcut-ref="privatePointcut" ➥
  method="doAfterThrowing" throwing="e"/>
  <aop:after pointcut-ref="privatePointcut" method="doAfter"/>
  <aop:around pointcut-ref="privatePointcut" method="doProfile"/>
  <aop:declare-parents types-matching="cn.spring21.unveilspring.target.*" ➥
  implement-interface="cn.spring21.unveilspring.introduction.ICounter" ➥
  default-impl="cn.spring21.unveilspring.introduction.CounterImpl"/>
</aop:aspect>
```

如我们所见，基于Schema的Introduction的定义是通过<aop:aspect>下的<aop:declare-parents>进行的：

- types-matching。用于指定要对哪些目标对象进行Introduction逻辑的织入。
- implement-interface。指定新增加的Introduction行为的接口定义类型。
- default-impl。指定新增加的Introduction行为的接口定义的默认实现类。

在将该配置加入容器的基于Schema的配置文件之后，所有cn.spring21.unveilspring.target包下的目标对象将被添加ICounter接口定义的行为。

4. 其他需要关注的地方

因为剩下几个话题涉及多个概念，所以，需要统一在本节中讲述一下。

- Advice的参数化

与@AspectJ形式的AOP一样，在基于Schema的AOP中，Aspect内声明的各个Advice对应的方法定义，可以接受指定类型的参数声明，只要按照规定的顺序声明参数即可。

在各个Advice对应的Pointcut定义中通过`args`、`@annotation`等标志符指定了要绑定的参数名称之后，我们就可以在相应的Advice方法上声明对应的参数。当Advice逻辑被调用时，这些参数的参数值将被绑定到方法调用。具体的信息可以参照本章前面的内容。

如果我们要明确这些参数名称，也可以通过各个Advice对应的Schema配置元素的`arg-names`属性明确指定。不过，通常情况下，只要按照顺序定义Pointcut和Advice方法的参数声明，就可以省去`arg-names`的使用。

注意 要了解更多Advice参数化相关信息，可参照Spring参考文档“6.3.3.6Advice parameters”一节。

- Advice的执行顺序

当多个Advice需要在同一个Joinpoint处执行时，多个Advice的执行顺序与之前在@AspectJ形式的AOP部分提及的内容相同，即同一个Aspect内声明的多个Advice，优先级由它们声明的顺序决定，最先声明的Advice拥有较高的优先级；位于多个不同Aspect内的Advice，它们的执行顺序由Aspect的顺序决定，具体语义以`Ordered`接口定义为准。

在基于Schema的AOP中，要指定Aspect的顺序号，可以通过`<aop:aspect>`的`order`属性进行，如下所示：

```
<aop:aspect id="..." ref="..." order="100">
   ...
</aop:aspect>
```

- Aspect的实例化模式

@AspectJ形式的AOP，除了支持默认的singleton模式的实例化，还支持perthis和pertarget实例化模式。不过，对于基于Schema的AOP来说，它的Aspect只支持singleton实例化模式。

10.3 小结

到目前为止，我们接触了如下三种Spring AOP的使用方式。

- Spring AOP 1.*x*版本发布的基于接口定义的Advice声明方式。这种方式的特点是，各种类型的Advice定义需要实现特定的接口，Advice的管理可以通过IoC容器或者直接编程来进行。该方式从Spring AOP发布以来一直工作得很好，我们将称其为第一代Spring AOP。
- @AspectJ形式的AOP是在Spring 2.0发布之后新增加的一种Spring AOP使用方式。在这种方式中，我们只需要以POJO的形式声明相应的Aspect和Advice，然后通过相应的注解标注一下即可。该方式的好处是，各种信息的管理统一到了一个位置，并且由于IDE的重构支持，AOP实现的管理更加方便高效。另外，只需要使用注解标注POJO中的Advice定义方法即可，而不用实现规定的接口来实现各种Advice类型。
- 基于Schema的AOP也是在Spring 2.0发布之后增加的Spring AOP使用方式。它融合了第一代Spring AOP和@AspectJ形式的AOP优点，不但完全支持第一代Spring AOP的配置需求，而且在

@AspectJ形式的AOP的基础之上进行了改装：我们依然可以通过POJO声明Aspect和Advice定义。不过，通过相应注解表达的各种信息，转移到了XSD形式的容器配置文件中。

对于如何在这三种方式之间进行选择，应该依照当前应用环境、各种工具支持、整体团队对各种方式的熟悉程度等不同情况来权衡利弊，最终决定一种合适的方式。如果先抛开其他限制，要我选择的话，我会选择@AspectJ形式的AOP，原因为如下几点。

(1) Aspect和Advice的声明完全基于最普通的POJO。

(2) 使用注解作为配置信息的载体，在当前IDE良好支持的大好前景下，重构不再是问题。

(3) AOP概念实体的声明和相关配置全部集中于一处管理，当你为了修改一点儿功能而要在多个文件之间"徜徉"的时候，就会发现将所有信息统一于一处管理的好处。

(4) 强类型的方法参数声明，再也不用直接操作`Object[]`，从而避免了信息的查找、对象的强制类型转换等琐事。

(5) 通过Aspect将公用的Pointcut定义模块化管理。

当然，在使用的过程中可能还会发现更多让人惊喜的东西。

不过，在实际开发过程中，并不是完全由我们的一厢情愿来决定使用哪种方式的。要使用@AspectJ形式的AOP，首先一点就是，我们的应用程序需要运行于Java 5以及更高版本的Java虚拟机之上。而现在许多应用程序还依赖于1.4.2甚至更低版本的JDK或者JRE。这时，可以选择使用基于Schema的AOP。

基于Schema的AOP允许我们使用POJO来声明相应的Aspect和Advice定义，但是不强求使用Java 5以上版本的Java虚拟机。除了一些细小功能上的限制外，基于Schema的AOP可以算是继承了@AspectJ形式的AOP的大部分优点。如果我们能够升级到Spring 2.*x*版本，但是却无法使用Java 5，那么基于Schema的AOP将是我们的最佳选择。基于Schema的AOP将AOP实体的实现和相应配置划分到两个不同的位置，在各种部件的管理上可能不是很方便，也无法获得重构的好处。不过，随着IDE对XML的支持越来越好，这些问题最终将得到解决，当使用Eclipse 3.3编辑Spring的配置文件时，已经可以在Java文件和XML之间跳转了。

随着2.*x*版本的Spring AOP发布，第一代Spring AOP好像可以退出历史舞台了。但实际上，它依然扮演着重要的角色。不要忘了，在2.*x*版本的AOP中，不论是对AspectJ集成还是基于Schema的AOP的实现，最终的底层实现还是基于第一代Spring AOP的各种理念和实体。虽然第一代Spring AOP不像2.*x*之后能够直接通过POJO声明Aspect和Advice，但是它的Advice实现依然拥有很强的表现力。在功能的实现方面，三种方式实际上是等效的。如果你一直使用第一代Spring AOP，那么熟悉它的使用将是你选择它的最好理由。不过，既然2.0发布后配置文件已经从DTD转向提倡的XSD形式，对于"第一代Spring AOP"的配置也不妨转向"基于Schema的AOP"的配置，即所谓"新瓶装旧酒"！

至此，Spring AOP框架的相关讲解就算基本完成了。对于它的使用和实现原理，我想大家已经心中有数了。

注意　Spring参考文档中有一部分内容专门介绍如何将AspectJ集成到Spring应用程序中。因为已经完全偏重于AspectJ和集成的话题，所以，我们在这里就不做介绍了。如果你的应用程序使用了AspectJ，同时还要使用Spring，那么在集成这两者的时候，可以参照Spring参考文档的这部分内容。

下一章，我们关注使用AOP的一些最佳实践。

第11章

AOP应用案例

本章内容

- 异常处理
- 安全检查
- 缓存

在结束有关Spring AOP的话题之前，有必要从最一般的层面上对适合使用AOP的场景做一简单的介绍。如果称这些场景为AOP的最佳实践，其实未尝不可！

11.1 异常处理

或许你已经在使用AOP的方式进行异常处理，但是可能并不知道这实际上对应着一个很有趣的术语，叫做Fault Barrier。同样地，在接触这个术语之前，我实际上已经将这个概念所阐述的理念应用于工作中，直到看到dev2dev网站上的一篇文章"Effective Java Exception"，才知道原来这种异常的处理方式还对应这么一个有趣的术语。不过，在讲述Fault Barrier之前，我们有必要先来回顾一下Java中异常处理的相关内容。

11.1.1 Java 异常处理

异常处理是个很大的话题，限于篇幅，我们不可能详细讲述异常处理的方方面面。下面只是简单地回顾一下Java中的异常类型和对应的处理方式，以便引出Fault Barrier的概念。

Java不是最早也不是唯一使用异常机制的语言。不过，Java却在引入Checked Exception的基础上为其自身的异常处理添加了少许的新意。我们先来看一下Java中的异常层次体系大体上是一种什么样的结构，如图11-1所示。

在图11-1中，我们将Java中的异常类型分为如下两类。

- 通常将`java.lang.Error`和`java.lang.RuntimeException`及其子类称之为unchecked exception。之所以如此称呼，是因为编译器不会对这些类型的异常进行编译期检查。因为`java.lang.Error`我们通常关心不着，所以，狭义上将`java.lang.RuntimeException`及其子类暂且称为unchecked exception也是可以的。
- `java.lang.Exception`及其子类，但除去`java.lang.RuntimeException`分支，统称为checked exception。一旦在程序的方法定义中声明了将会抛出"checked exception"，调用程序就必须对这些异常进行处理，而且编译器会在编译期间对这些异常类型进行检查。

各位大师级人物对checked exception和unchecked exception的论战暂且放在一边，我们主要讲述一下二者具体的应用场景。

- unchecked exception通常对应系统中的严重异常情况，这些情况应用程序一般无法恢复，比如数据库挂掉、网线连接中断、服务器崩溃等。所以，unchecked exception所提供的信息一般不是为应用程序准备的，而是为人准备的，确切地说，是为了能够让系统维护人员能够根据所提供的信息来判定到底哪里出了问题，以便人工干预。
- checked exception引入Java后，一直是备受争议。不过，我觉得这与概念本身没有任何关系，是否被应用于合适的场合是由人来决定的，而不是概念本身。checked exception通常用于表明系统中的某些罕见的非正常状态。对于一个业务方法来说，使用错误号（Error Code）的时代是通过返回-1之类数字表明一些非正常状态，现在可以通过抛出不同类型的checked exception来表明这些非正常状态，并要求调用方对这些非正常状态进行处理，而编译器对checked exception的检查可以进一步加强这种契约关系。通常checked exception是可恢复的，也是意料之中的，它所提供的信息是面向应用程序的，应用程序可以根据系统逻辑对不同的checked exception类型进行有针对性的处理。

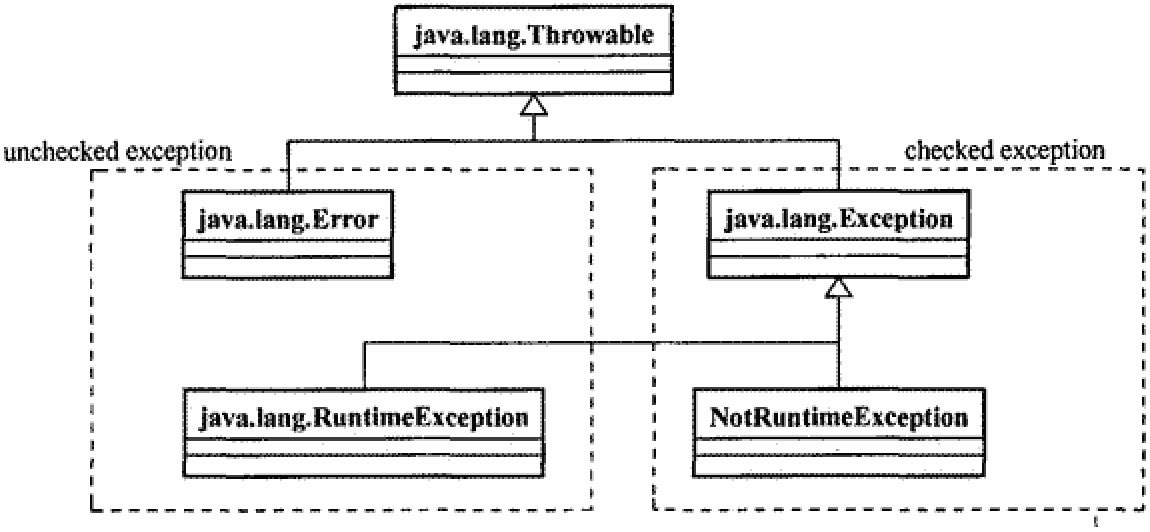

图11-1 Java异常层次体系简图

在技术文章“Effective Java Exception”中，作者将unchecked exception对应的情况称之为Fault，而将checked exception对应的情况称之为Contingency。而Fault Barrier要处理的，就是对应Fault的情况，即unchecked exception。

11.1.2 Fault Barrier

对于checked exception来说，不同的类型可以有不同的处理方式，这完全是由系统逻辑来决定的，调用方可以根据不同的类型，有针对性地对checked exception进行处理。反过来，对于unchecked exception来说，不同的类型则是没有太多必要的，因为不管你的应用程序抛出何种类型的unchecked exception，最终都是需要人来进行干预，只要unchecked exception能够提供足够的信息，相应人员就可以进行处理，几乎就是无差别对待。

当系统中多个地方都可能抛出unchecked exception的时候，在引入Fault Barrier概念之前，我们可能会在每个调用的最顶层，分别添加异常处理逻辑对其进行处理。而就像前面所说的那样，unchecked exception实际上可以做的事情很少，通常就是记录日志、通知相应人员。所以，这些相同的逻辑实现可以归并于一处进行处理，而不是让它们散落到系统的各处，也就是说，对于系统中的Fault来说，它实际上就是一种横切关注点（cross-cutting concern）。

鉴于此，我们完全可以实现一个对应Fault处理的Aspect，让其对系统中的所有可能的Fault情况进行统一的处理。这个专职于处理Fault的Aspect，我们就称之为Fault Barrier。

实际上，我们从讲解Spring AOP的ThrowsAdvice开始，就提供了一个Fault Barrier的实现实例。在该实例中，我们通过电子邮件的方式将系统中的Fault情况，也就是以unchecked exception形式给出的信息转发给相关人员，并记录到日志。当然，如果可能，你还可以加入更多的处理，比如分析unchecked exception信息、为相关人员提供更加友好的系统信息等。

注意 任何的概念都会有特例，对于异常来说也是如此，比如unchecked exception也可以被相应的方法声明并被调用方捕获。不过，前面也说了，篇幅有限，无法完全囊括所有内容。特殊的情况，如果后继内容有所涉及，我们会给予相应解释。

11.2 安全检查

如果你已经使用Java开发Web应用程序多年，那么一定不会对`Filter`感到陌生吧？`javax.servlet.Filter`是Servlet规范为我们提供的一种AOP支持，通过它，我们可以为基于Servlet的Web应用添加相应的资源访问控制（当然，还可以做很多其他事情）。不过，基于`Filter`的Web应用的资源访问控制，仅仅是特定领域的安全检查需求。实际上，通过AOP，我们可以为任何类型的应用添加相应的安全支持。

在介绍AOP概念的时候就曾经提到过，安全检查属于系统的一种横切关注点，按照原先的方法进行系统开发，势必让这些安全检查逻辑散落系统各处，所以，对付它的最好办法就是用AOP。在将系统中可能需要安全检查的点排查清楚之后，我们就可以为这些点织入安全检查的逻辑了。

要为系统中某个点添加安全支持，最简单的办法就是提供一个拦截器，对所有访问该点的调用进行拦截。所以，对于基本的安全检查的Aspect实现来说，如代码清单11-1所示。

代码清单11-1 用于安全检查的Aspect定义示例

```
@Aspect
public class SecurityAspect
{
  @Around("...")
  public Object doCheck(ProceedingJoinPoint pjp) throws Throwable
  {
    if(isIllegalRequest(pjp))
    {
      throw new SecurityCheckingException("necessary information");
    }
    return pjp.proceed();
  }
}
```

既然我们崇尚“不重新发明轮子”，在动手之前，有必要使用Google搜索一下是否有现成的，免去人力物力的浪费。

实际上，作为基于Spring平台的一套安全框架，Acegi框架（也就是最新的Spring Security）已经可以说在企业级应用的安全领域声名远扬了。它在Spring基础上，提供了完备的系统认证、授权、访问控制等安全检查功能。Acegi框架最初是独立于Spring开发的，现在已经并入Spring Portfolio，更名为Spring Security，可以在http://www.acegisecurity.org/获得有关Acegi的更多信息。

11

11.3 缓存

AOP应用的另一个主要场景是为系统透明地添加缓存支持。缓存可以在很大程度上提升系统的性能，但它不是业务需求，而是系统需求。在现有方法论的基础上为系统添加缓存支持，就会因为系统中缓存需求的广泛分布，造成实现上的代码散落。

为了避免需要添加的缓存实现逻辑影响业务逻辑的实现，我们可以让缓存的实现独立于业务对象的实现之外，将系统中的缓存需求通过AOP的Aspect进行封装，只在系统中某个点确切需要缓存支持的情况下，才为其织入。

使用AOP为系统添加缓存很简单，如代码清单11-2所示。（代码示例没有添加同步逻辑，在生产环境下实现类似功能的时候需要考虑到这一点。）

代码清单11-2 用于缓存的Aspect原型示例

```
@Aspect
public class CachingAspect
{
  private static Map cache = new LRUMap(5);

  @Around("...")
  public Object doCache(ProceedingJoinPoint pjp,Object key) throws Throwable
  {
    if(cache.containsKey(key))
    {
      return cache.get(key);
    }
    else
    {
      Object retValue = pjp.proceed();
      cache.put(key, retValue);
      return retValue;
    }
  }
}
```

在没有使用AOP之前，要为系统某个地方加入缓存的话，恐怕也是以差不多的逻辑实现的。

不过，现在不需要这么做了，原因如下。

- 现在已经有许多现成的Caching产品实现，如EhCache、JBossCache等。
- Spring Modules项目提供了对现有Caching产品的集成，这样就可以通过外部声明的方式为系统中的Joinpoint添加Caching支持。

可以从https://springmodules.dev.java.net/获得更多有关Spring Modules项目的信息。另外，dev2dev网站有一篇文章“Declarative Caching Services for Spring”[①]专门介绍了如何通过Spring Cache为应用程序添加声明性的Caching支持。

11.4 小结

本章给出了几种常见的AOP应用场景（或最佳实践），以帮助读者扩展AOP的应用思路。但实际上，AOP的应用场景可以很广，而不只局限于本章罗列的这几种情况。关于AOP的更多应用场景和最佳实践，需要读者在实际开发中自己去挖掘，去探索。

---

① http://dev2dev.bea.com/pub/a/2006/05/declarative-caching.html。

第12章

Spring AOP之扩展篇

本章内容

- 有关公开当前调用的代理对象的探讨

12.1 有关公开当前调用的代理对象的探讨

这个话题应该从同一个对象内的嵌套方法调用拦截失效说起。

12.1.1 问题的现象

假设我们有如代码清单12-1所示的目标对象类定义。当然，这样的类定义可以映射到系统中的任何可能的业务对象。

代码清单12-1 同一对象内方法嵌套调用的目标对象示例

```
public class NestableInvocationBO {

  public void method1()
  {
          method2();
          System.out.println("method1 executed!");
  }

  public void method2()
  {
    System.out.println("method2 executed.");
  }
}
```

该类定义中需要我们关注的是它的某个方法会调用同一对象上定义的其他方法。这通常是比较常见的。在NestableInvocationBO类中，method1方法调用了同一对象的method2方法。

现在，我们要使用Spring AOP拦截该类定义的method1和method2方法，比如加入一个简单的性能检测逻辑。那么可以定义一个PerformanceTraceAspect，如代码清单12-2所示。

代码清单12-2 用于性能检查的Aspect定义

```
@Aspect
public class PerformanceTraceAspect {
  private final Log logger = LogFactory.getLog(PerformanceTraceAspect.class);

  @Pointcut("execution(public void *.method1())")
  public void method1(){}
  @Pointcut("execution(public void *.method2())")
```

```
    public void method2(){}
    @Pointcut("method1()||method2()")
    public void compositePointcut(){};

    @Around("compositePointcut()")
    public Object performanceTrace(ProceedingJoinPoint joinpoint) throws Throwable
    {
      StopWatch watch = new StopWatch();
      try
      {
        watch.start();
        return joinpoint.proceed();
      }
      finally
      {
        watch.stop();

        if(logger.isInfoEnabled())
          logger.info("PT in method["+ ➥
          joinpoint.getSignature().getName()+➥
          "]>>>>>"+watch.toString());
      }
    }
}
```

我们的Around Advice定义会拦截compositePointcut()所指定的Joinpoint，即method1或者method2的执行。

如果将PerformanceTraceAspect中定义的横切逻辑织入NestableInvocationBO，然后运行如下代码并查看结果：

```
AspectJProxyFactory weaver = new AspectJProxyFactory(new NestableInvocationBO());
weaver.setProxyTargetClass(true);
weaver.addAspect(PerformanceTraceAspect.class);
Object proxy = weaver.getProxy();
((NestableInvocationBO)proxy).method2();
((NestableInvocationBO)proxy).method1();
```

会得到如下的输出结果：

```
method2 executed!
701 [main] INFO ...PerformanceTraceAspect  - PT in method[method2]>>>>>0:00:00.000
method2 executed!
method1 executed!
701 [main] INFO ...PerformanceTraceAspect  - PT in method[method1]>>>>>0:00:00.000
```

发现问题没有？当我们从外部直接调用NestableInvocationBO对象的method2的时候，因为该方法签名匹配PerformanceTraceAspect中Around Advice所对应的Pointcut定义，所以，Around Advice逻辑得以执行，也就是说，PerformanceTraceAspect拦截method2的执行成功了。但是，当调用method1的时候，却只有method1方法的执行拦截成功，而method1方法内部的method2方法执行却没有被拦截，因为输出日志中只有PT in method[method1]的信息。

12.1.2 原因的分析

这种结果的出现，归根结底是由Spring AOP的实现机制造成的。我们知道，Spring AOP采用代理模式实现AOP，具体的横切逻辑会被添加到动态生成的代理对象中，只要调用的是目标对象的代理对象上的方法，通常就可以保证目标对象上的方法执行可以被拦截。就像NestableInvocationBO的

method2方法执行一样，当我们调用代理对象上的method2的时候，目标对象的method2就会被成功拦截。

不过，代理模式的实现机制在处理方法调用的时序方面，会给使用这种机制实现的AOP产品造成一个小小的"缺憾"。我们来看一般的代理对象方法与目标对象方法的调用时序，如下所示：

```
proxy.method2
{
    记录方法调用开始时间;
    target.method2;
    记录方法调用结束时间;
    计算消耗的时间并记录到日志;
}
```

在代理对象方法中，不管你如何添加横切逻辑，也不管你添加多少横切逻辑，有一点是确定的，那就是，你终归需要调用目标对象上的同一方法来执行最初所定义的方法逻辑。

如果目标对象中原始方法调用依赖于其他对象，那没问题，我们可以为目标对象注入所依赖对象的代理，并且可以保证相应Joinpoint被拦截并织入横切逻辑。而一旦目标对象中的原始方法调用直接调用自身方法的时候，也就是说，它依赖于自身所定义的其他方法的时候，问题就来了，如图12-1所示。

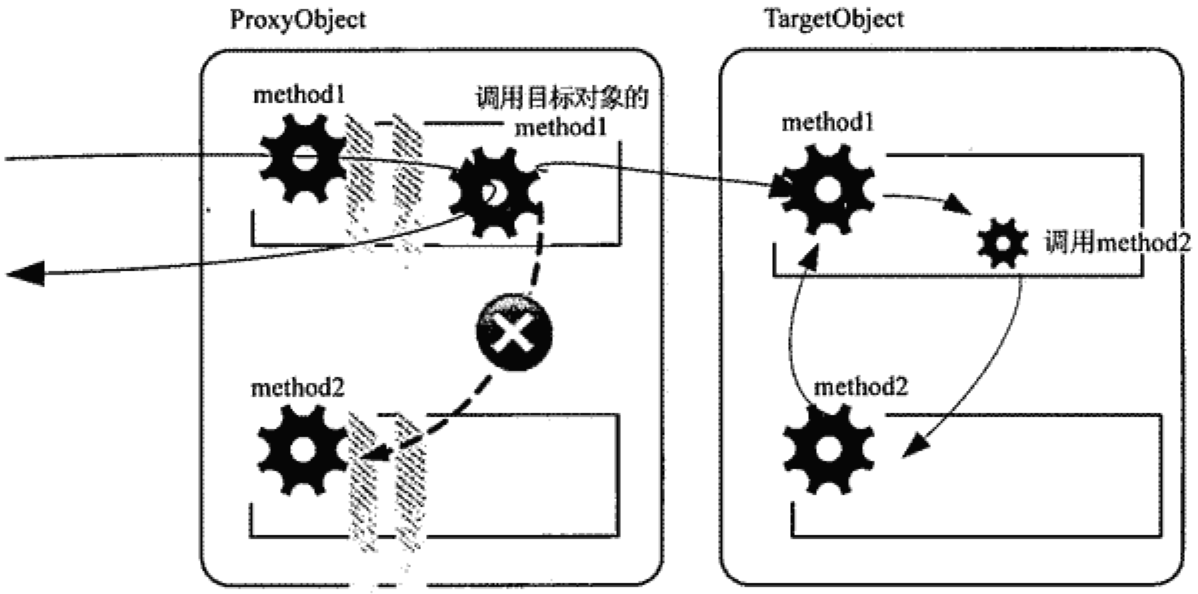

图12-1 同一对象内部方法嵌套调用示意图

在代理对象的method1方法执行经历了层层拦截器之后，最终会将调用转向目标对象上的method1，之后的调用流程全部都是走在TargetObject之上，当method1调用method2时，它调用的是TargetObject上的method2，而不是ProxyObject上的method2。要知道，针对method2的横切逻辑，只织入到了ProxyObject上的method2方法中，所以，在method1中所调用的method2没有能够被成功拦截。

12.1.3 解决方案

知道了原因，我们就可以"对症下药"了。

当目标对象依赖于其他对象时，我们可以通过为目标对象注入依赖对象的代理对象，来解决相应的拦截问题。那么，当目标对象依赖于自身时，我们也可以尝试将目标对象的代理对象公开给它，只

要让目标对象调用自身代理对象上的相应方法，就可以解决内部调用的方法没有被拦截的问题。

Spring AOP提供了AopContext来公开当前目标对象的代理对象，我们只要在目标对象中使用AopContext.currentProxy()就可以取得当前目标对象所对应的代理对象。现在，我们重构目标对象，让它直接调用它的代理对象的相应方法，如代码清单12-3所示。

代码清单12-3　使用AopContext获得当前目标对象的代理对象的代码示例

```
public class NestableInvocationBO {

   public void method1()
   {
      ((NestableInvocationBO)AopContext.currentProxy()).method2();
      System.out.println("method1 executed!");
   }

   public void method2()
   {
      System.out.println("method2 executed!");
   }
}
```

要使AopContext.currentProxy()生效，我们在生成目标对象的代理对象时，需要将ProxyConfig或者它的相应子类的exposeProxy属性设置为true，如下所示：

```
AspectJProxyFactory weaver = new AspectJProxyFactory(new NestableInvocationBO());
weaver.setProxyTargetClass(true);
weaver.setExposeProxy(true);
weaver.addAspect(PerformanceTraceAspect.class);
Object proxy = weaver.getProxy();
((NestableInvocationBO)proxy).method2();
((NestableInvocationBO)proxy).method1();
```

现在，我们得到了想要的拦截结果：

```
method2 executed!
611 [main] INFO ...PerformanceTraceAspect  - PT in method[method2]>>>>>0:00:00.000
method2 executed!
611 [main] INFO ...PerformanceTraceAspect  - PT in method[method2]>>>>>0:00:00.000
method1 executed!
611 [main] INFO ...PerformanceTraceAspect  - PT in method[method1]>>>>>0:00:00.000
```

任何匹配Pointcut定义的method2都得到了成功拦截。

问题是解决了，但解决的方式不是很雅观，因为我们的目标对象都直接绑定到了Spring AOP的具体API上了。所以，我们考虑重构目标对象定义。既然我们知道能够通过AopContext. currentProxy()取得当前目标对象对应的代理对象，那完全可以在目标对象中声明对其代理对象的依赖，而由IoC容器来帮助我们注入这个代理对象。

注入的方式可以不一而足。

- 可以在目标对象中声明一个实例变量作为其代理对象的引用，然后由构造方法注入或者setter方法注入将AopContext.currentProxy()取得的Object注入给这个声明的实例变量。
- 在目标对象中声明一个getter方法，如getThis()，然后通过Spring的IoC容器的方法注入或者方法替换，将这个方法的逻辑替换为return AopContext.currentProxy()。这样，在调用自身方法的时候，直接通过getThis().method2();就可以了。
- 声明一个Wrapper类，并且让目标对象依赖于这个类。在Wrapper类中直接声明一个getProxy()

或者类似的方法，将`return AopContext.currentProxy()`类似逻辑添加到这个方法中，目标对象只需要`getWrapper().getProxy()`就可以取得相应的代理对象。`Wrapper`类分离了目标对象与Spring API的直接耦合。至于让这个`Wrapper`以`Utility`类出现，还是在目标对象中直接构造，或者依赖注入到目标对象，这就由你来定了。

- 为类似的目标对象声明统一的接口定义，然后通过`BeanPostProcessor`处理这些接口实现类，将实现类的某个取得当前对象的代理对象的方法逻辑覆盖掉。这与方法替换所使用的原理一样，只不过可以借助Spring的IoC容器进行批量处理而已。

至于其他更好的方式，还是留待读者自己去考虑吧，我就不啰嗦了。

> **注意** 实际上，这种情况的出现仅仅是因为Spring AOP采用的是代理机制实现。如果像AspectJ那样，直接将横切逻辑织入目标对象，那么代理对象和目标对象实际上就合为一体了，调用也不会出现这样的问题。如果你觉得以上处理方式不好接受，那么改用AspectJ也无妨。不过，使用中同样会遇到这样或那样的问题，这简直是一定的。我的建议是用自己感觉合适的产品或者做事方式。

12.2 小结

在本部分内容结束之前，我们在本章中揭示了Spring AOP实现机制导致的一个小小的陷阱，并通过分析问题产生的原因给出了自己的解决方案。同时，也在某些方面给出了适当的扩展讨论。

应该说，Spring AOP作为一个轻量级的AOP框架，在简单与强大之间取得了很好的平衡。合理地使用Spring AOP，将帮助我们更快更好地完成各种工作。最后，希望大家在Spring AOP的使用之路上愉快地前行。

Part 4 第四部分

使用 Spring 访问数据

为了简化 Java 应用程序开发，Spring 框架提供了数据访问层。数据访问层主要可以划分为如下三个部分。

- **统一的数据访问异常层次体系（Consistent Exception Hierarchy）**。Spring 框架将特定的数据访问技术相关的异常（Exception）进行转译，然后封装为一套标准的异常层次体系。通过这套标准异常层次体系，不管使用的数据访问技术如何变化，客户端对象只需要捕获并处理这套标准的异常就可以，再也不需要因为所使用的数据访问技术变更或者迁移等问题，而对自身代码做任何改动。
- **JDBC（Java DataBase Connectivity，Java 数据库连接）API 的最佳实践**。JDBC 作为一套数据访问标准是很成功的，它规范了各个数据库厂商之间的数据访问接口，极大地促进了 RDBMS（Relational DataBase Management System，关系数据库管理系统）在 Java 平台上的迅速普及。不过，任何事物都有瑕疵，虽然作为一套标准 JDBC 很成功，但在其 API 的设计和使用上则不尽然，如下所述。
  - SQLException 的设计本身没有将自身作为标准的职责履行到底，各种具体的异常信息全部让各个 RDBMS 厂商自行界定，从而导致应用程序需要根据具体的数据库提供商来判定异常中所提供的信息到底是什么意思。
  - JDBC API 较为贴近底层，使用上比较烦琐。如果不做适当的封装，在使用上很容易造成问题。如数据库连接没有释放，就是最容易看到的情况。

Spring 框架针对 JDBC API 使用上的种种问题，提出了一套最佳实践。

- **以统一的方式对各种 ORM 方案的集成**。除了使用标准的 JDBC 进行数据库的访问，现在使用比较多的就是 ORM（Object Relational Mapping，对象-关系映射，主要用来屏蔽对象与关系数据库之间结构的非一致性。大部分的 ORM API 在使用上与 JDBC API 非常相似，所以，Spring 也以与 JDBC API 的最佳实践同样的方式，集成了现有的各种 ORM 方案。同时，也将这些 ORM 特定的异常纳入了它的统一的异常层次体系。

总地来说，Spring 的数据访问层实际上就是以统一的数据访问异常层次体系为核心，以 JDBC API 的最佳实践和统一集成各种 ORM 方案为两翼，为 Java 平台的数据访问铺就了一条平坦的大道。

现在，让我们先从统一的数据访问异常层次体系开始，探索一下 Spring 的数据访问层到底蕴含什么奥妙吧！

本部分内容

- 第 13 章　统一的数据访问异常层次体系
- 第 14 章　JDBC API 的最佳实践
- 第 15 章　Spring 对各种 ORM 的集成
- 第 16 章　Spring 数据访问之扩展篇

第13章

统一的数据访问异常层次体系

本章内容

- DAO模式的背景
- 梦想照进现实
- 发现问题，解决问题
- 不重新发明轮子

要了解Spring为什么要提供统一的数据访问异常层次体系，我们得先从DAO模式说起。

13.1 DAO 模式的背景

不管是一个逻辑简单的小软件系统，还是一个关系复杂的大软件系统，都可能涉及对相关数据的访问和存储，而这些数据的存储机制和访问方式往往随场景不同而各异。为了统一和简化相关的数据访问操作，J2EE核心模式提出了DAO（Data Access Object，数据访问对象）模式。使用DAO模式，可以完全分离数据的访问和存储，很好地屏蔽了各种数据访问方式的差异性。不论数据存储在普通的文本文件或者csv文件，还是关系数据库（RDBMS）或者LDAP（Lightweight Directory Access Protocol，轻量级目录访问协议）系统中，使用DAO模式访问数据的客户端代码可以完全忽视这种差异，而以统一的接口来访问相应数据。

空话多说无益，我们还是来看一个具体的应用DAO模式的场景吧！

对于大部分软件系统来说，访问顾客信息是大家经常接触的。我们就以访问顾客信息为例，看一下如何使用DAO模式。使用DAO模式访问顾客数据，首先需要声明一个数据访问对象接口定义，如下代码所示：

```
public interface ICustomerDao {
  Customer findCustomerByPK(String customerId);
  void     updateCustomerStatus(Customer customer);
  //...
}
```

对于客户端代码来说，如通常的服务层代码（见代码清单13-1），只需要声明依赖该数据访问接口即可，所有的数据访问全部通过该接口进行。即使以后因为数据存储机制发生变化，而导致DAO接口的实现类发生变化，客户端代码也不需要做任何的调整（当然，如果设计不良，依然会存在需要调整客户端代码的情况）。

代码清单13-1　依赖DAO接口的CustomerService类定义

```
public class CustomerService
{
```

```
    private ICustomerDao customerDao;

    public void disableCustomerCampain(String customerId)
    {
      Customer customer =
          getCustomerDao().findCustomerByPK(customerId);
      customer.setCampainStatus(CampainStatus.DISABLE);
      getCustomerDao().updateCustomerStatus(customer);
    }

    public ICustomerDao getCustomerDao() {
      return customerDao;
    }
    public void setCustomerDao(ICustomerDao customerDao) {
      this.customerDao = customerDao;
    }
}
```

通常情况下，顾客信息存储于关系数据库中，所以，相应地，我们会提供一个基于JDBC的DAO接口实现类，如代码清单13-2所示。

代码清单13-2　基于JDBC的ICustomerDao实现类定义

```
public class JdbcCustomerDao implements ICustomerDao
{
    public Customer findCustomerByPK(String customerId) {
      // TODO Auto-generated method stub
      return null;
    }
    public void updateCustomerStatus(Customer customer) {
      // TODO Auto-generated method stub
    }
}
```

可能随着系统需求的变更，顾客信息需要转移到LDAP服务，或者转而使用其他企业位于LDAP服务中的顾客信息，又或者别人要使用我们的CustomerService，但他们使用不同的数据存储机制，这时，就需要提供一个基于LDAP的数据访问对象，如代码清单13-3所示。

代码清单13-3　基于LDAP的ICustomerDao实现类定义

```
public class LdapCustomerDao implements ICustomerDao
{
    public Customer findCustomerByPK(String customerId) {
      // TODO Auto-generated method stub
      return null;
    }
    public void updateCustomerStatus(Customer customer) {
      // TODO Auto-generated method stub
    }
}
```

即使数据访问接口的实现类随着需求而发生变化，客户端代码（这里是CustomerService）也可以完全忽视这种变化，唯一需要变动的地方可能只是Factory对象的几行代码，甚至只是IoC容器配置文件中简单的class类型替换而已，而客户端代码无需任何变动。所以，DAO模式对屏蔽不同数据访问机制的差异性起到举足轻重的作用。

13.2 梦想照进现实

为了简化概念的描述，前面针对DAO的例子省略了部分细节，比如接口与实现之间的某种依赖性。不管是JdbcCustomerDao还是LdapCustomerDao，都省略了最基本的东西，即数据访问机制特定的代码。当引入这些特定于数据访问机制的代码时，问题就产生了，最明显的莫过于特定于数据访问机制的异常处理。

当把具体的JDBC代码充实到JdbcCustomerDao时（见代码清单13-4），看看哪里不对头。

代码清单13-4　添加了具体JDBC代码后的JdbcCustomerDao的定义

```
public Customer findCustomerByPK(String customerId)
{
  Connection con = null;
  try {
    con = getDataSource().getConnection();
    //...
    Customer cust = ...;

    return cust;
  }
  catch (SQLException e) {
    // 是抛出呢？还是在当前位置处理呢？
  }
  finally {
    releaseConnection(con);
  }
}

private void releaseConnection(Connection con) {
  // TODO Auto-generated method stub
}
```

使用JDBC进行数据访问，当其间出现问题的时候，JDBC API会抛出SQLException来表明问题的发生。而SQLException属于checked exception，所以，我们的DAO实现类需要捕获这种异常并进行处理。

那么，该如何处理DAO实现类中捕获的SQLException呢？直接在DAO实现类处理掉？可是这样的话，客户端代码如何得知在数据访问期间发生了问题？所以，只好先直接将SQLException抛给客户端，进而，DAO实现类的相应方法签名也需要修正为抛出SQLException，如下所示：

```
public Customer findCustomerByPK(String customerId) throws SQLException{
  ...
}
```

相应地，DAO接口定义中的相应方法签名也需要做如下修改：

```
public interface ICustomerDao {
  Customer findCustomerByPK(String customerId) throws SQLException;
  void    updateCustomerStatus(Customer customer);
  //...
}
```

可是，这样就解决问题了吗？

- 问题1

我们的数据访问接口对于客户端来说是通用的，不管数据访问对象因为数据访问机制的不同而如何变更，客户端代码不应该受其牵连。 但是，现在因为使用JDBC做数据访问，需要抛出特定的SQLException，那么客户端代码就需要捕捉该异常并做相应的处理。这是与数据访问对象模式的设计初衷相背离的。

- 问题2

在引入另一种数据访问机制时，问题更是接踵而来。当加入LdapCustomerDao实现时，LdapCustomerDao需要抛出NamingException。如果要保证findCustomerByPK方法实现了ICustomerDao中的方法，那么就得更改ICustomerDao的方法签名，如下所示：

```
Customer findCustomerByPK(String customerId) throws SQLException,NamingException;
```

糟糕不是吗？我们又把统一的访问接口给改了，相应的客户端代码又要捕捉NamingException做相应的处理。如果随着不同数据访问对象实现的增多，以及考虑数据访问对象中其他数据访问方法，这种糟糕的情况不得继续下去吗？

也就是说，因为数据访问机制的不同，我们的数据访问接口的定义现在变成了空中楼阁，我们无法最终定义并确定这个接口，不是吗？

13.3 发现问题，解决问题

问题出现了，我们就应该尝试解决问题，因为我们实在不忍舍弃DAO模式所描述的那幅场景。那么如何来避免以上问题呢？

(1) 既然直接在DAO实现类内部处理SQLException这条路走不通，而将SQLException直接抛出又不可行，那么将SQLException或者其他特定的数据访问异常进行封装后再抛出又会如何？ 如果要这么做，以什么类型的异常进行封装然后再抛出呢？是checked exception还是unchecked exception？

大部分的或者说所有的数据访问操作抛出的异常对于客户端来说是系统的Fault，客户端是无法有效处理的，比如数据库操作失败、无法取得相应资源等。客户端对这些情况最有效的处理方式就是不做处理。因为客户端代码对于系统的Fault通常无法处理（当然如果必要，捕捉后处理也是可以的，比如捕捉相应异常后重试等），所以，将SQLException以及其他特定于数据访问机制的异常，以unchecked exception进行封装然后抛出，是比较合适的。

因为unchecked exception不需要编译器检查，ICustomerDao的数据访问方法就可恢复其本来面目而实现“大同”，如下所示：

```
Customer findCustomerByPK(String customerId)
```

各个DAO实现类内部，只要将SQLException及其他特定的数据访问异常，以unchecked exception进行封装即可，如代码清单13-5所示。

代码清单13-5 SQLException初步封装为unchecked exception的代码示例

```
public Customer findCustomerByPK(String customerId) {
  Connection con = null;
  try{
    con = getDataSource().getConnection();
    //...
    Customer cust = ...;

    return cust;
  }
```

```
    catch (SQLException e) {
      throw new RuntimeException(e);
    }
    finally{
      releaseConnection(con);
    }
  }
```

现在，统一数据访问接口定义的问题解决了。

(2) 以RuntimeException形式将特定的数据访问异常转换后抛出，虽然解决了统一数据访问接口的问题，但是，该方案依然不够周全。以SQLException为例，各个数据库提供商通过SQLException表达具体的错误信息时，所采用的方式是不同的，比如，有的数据库提供商采用SQLException的ErrorCode作为具体的错误信息标准，有的数据库提供商则通过SQLException的SqlState来返回详细的错误信息。即使将SQLException封装后抛出给客户端对象，当客户端对象需要了解具体的错误信息时，依然需要根据数据库提供商的不同，采取不同的信息提取方式。要知道，将这种错误信息的具体处理分散到各个客户端对象中处理是何等的糟糕？我们应该向客户端对象屏蔽这种差异性！

那么，如何来屏蔽这种差异性呢？答案当然是异常的分类转译（Exception Translation）。

a) 首先，不应该将对特定的数据访问异常的错误信息提取工作留给客户端对象，而是应该由DAO实现类，或者某个工具类以统一的方式进行处理。 我们暂且让具体的DAO实现类来做这个工作，那么对于JdbcCustomerDao来说，捕获异常后的处理就如代码清单13-6所示。

代码清单13-6 SQLException的异常转译伪代码示例

```
try
{
  //...
}
catch (SQLException e)
{
  if(isMysqlVendor())
  {
    // 按照Mysql数据库的规则分析错误信息(e)然后抛出
    throw new RuntimeException(e);
  }
  if(isOracleVendor())
  {
    // 按照Oracle数据库的规则分析错误信息(e)然后抛出
    throw new RuntimeException(e);
  }
  ...
}
```

b) 信息是提取出来了，可是，只通过RuntimeException一个异常类型，还不足以区分不同的错误类型，我们需要将数据访问期间发生的错误进行分类，然后为具体的错误分类，分配一个对应的异常类型。比如，数据库连接不上、ldap服务器连接失败，我们认为它们同属于获取资源失败；而主键冲突或者其他资源冲突，我们认为它们属于数据一致性冲突。那么，针对这些情况，就可以以RuntimeException为基准，为获取资源失败这种情况分配一个RuntimeException的子类型，称其为ResourceFailureException，而数据一致性冲突则可以对应RuntimeException的另一个子类型DataIntegrityViolationException，其他的分类和异常类型以此类推，这样，就有了如代码清单13-7所示的异常处理逻辑。

代码清单13-7　细化异常转译类型的代码示例

```
try
{
  //...
}
catch (SQLException e)
{
  if(isMysqlVendor())
  {
    if(1==e.getErrorCode())
      throw new ResourceFailureException(e);
    else if(1062 == e.getErrorCode())
      throw new DataIntegrityViolationException(e);
    else
      ...;
  }
  if(isOracleVendor())
  {
    int[] resourceFailureCodes = {17002,17447};
    int[] dataIntegrationViolationCode = {1,1400,1722,2291};
    ...
    if(ArrayUtils.contains(resourceFailureCodes,e.getErrorCode()))
      throw new ResourceFailureException(e);
    else if(ArrayUtils.contains(dataIntegrationViolationCode,e.getErrorCode()))
      throw new DataIntegrityViolationException(e);
    else
      ...;
  }
  ...
}
```

不论采用的是什么数据库服务器，也不论采用的是什么数据访问方式，除了示例所提到的基于JDBC的数据访问方式，还有对于其他的数据访问方式，只要将它们自身的异常，通过某种方式转译为以上提到的这几种异常类型，对于客户端对象来说，只需要关注这几种类型的异常，就可以知道到底出了什么问题，甚至系统监控人员也可以直接根据日志信息判断问题之所在。

说到底，在一套语义完整的异常体系定义完成之后，不管数据访问方式如何变换，只要相应的数据访问方式能够将自身的异常，转译到这套语义完整的异常体系定义之内，对于客户端对象来说，自身的数据访问异常处理逻辑从此就是岿然不动的。

实际上，我们需要的只是一套unchecked exception类型的面向数据访问领域的异常层次体系。

13.4　不重新发明轮子

现在，我们知道了unchecked exception类型的面向数据访问领域的异常层次体系存在的必要性，马上着手设计和实现它？不，我们已经有了现成的“轮子”啦！那就是Spring提供的数据访问异常层次体系。

Spring框架中统一的异常层次体系所涉及的大部分异常类型都定义在org.springframework.dao包中，处于这个体系的所有异常类型均以org.springframework.dao.DataAccessException为“统领”，然后根据职能划分为不同的异常子类型，总体上看，整个的异常层次体系如图13-1所示。

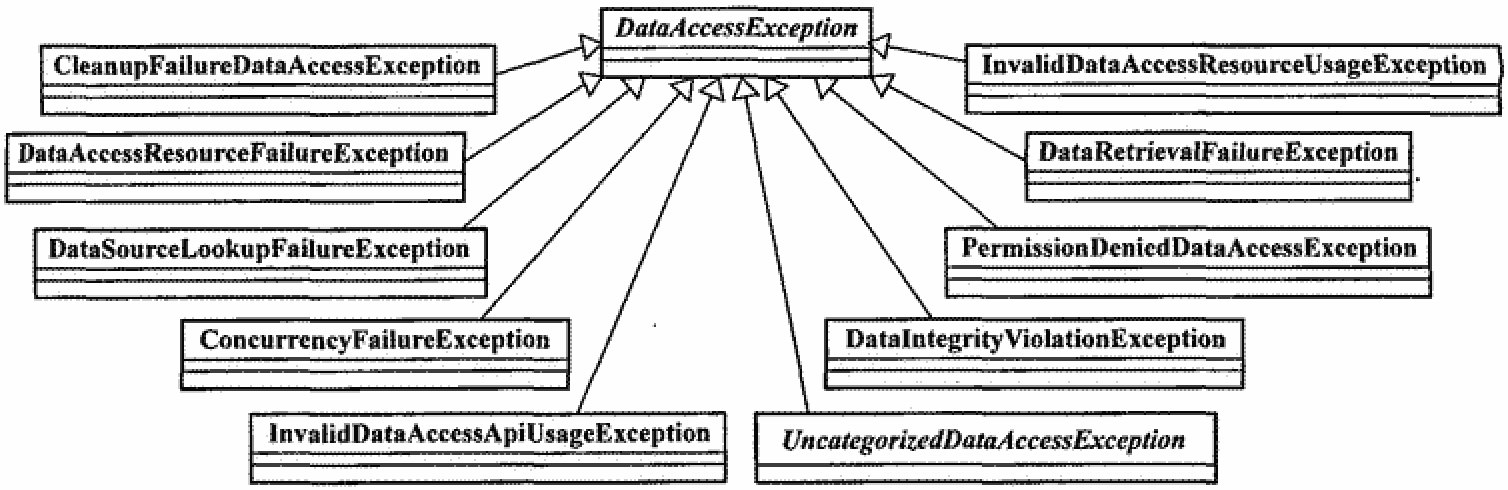

图13-1 异常层次体系简图

下面我们来看位于这个异常层次体系的各种异常类型定义，其具体职责是什么。

CleanupFailureDataAccessException。当已经成功完成相应的数据访问操作，要对使用的资源进行清理却失败的时候，将抛出该异常。比如，使用JDBC进行数据访问的时候，查询或者更新数据操作完成之后，需要关闭相应的数据库连接，如果在关闭连接的过程中出现SQLException，那么数据库连接没有被释放，导致资源清理失败。

DataAccessResourceFailureException。在无法访问相应的数据资源的情况下，将抛出DataAccessResourceFailureException。对应这种异常出现的最常见的场景就是数据库服务器挂掉的情况，这时，连接数据库的应用程序可以通过捕获该异常了解到是数据库服务器端出现了问题。对于JDBC来说，在数据库服务器挂掉之后，会抛出DataAccessResourceFailureException针对JDBC的具体子类型，即org.springframework.jdbc.CannotGetJdbcConnectionException。

DataSourceLookupFailureException。当尝试对JNDI（Java Naming and Directory Interface，Java命名和目录接口）服务上或者其他位置上的DataSource进行查找，而查找失败的时候，可以抛出DataSourceLookupFailureException。

ConcurrencyFailureException。并发进行数据访问操作失败的时候，可以抛出ConcurrencyFailureException，比如无法取得相应的数据库锁，或者乐观锁更新冲突等情况。根据不同的并发数据访问失败的情况，ConcurrencyFailureException细分为多个子类型，如图13-2所示。

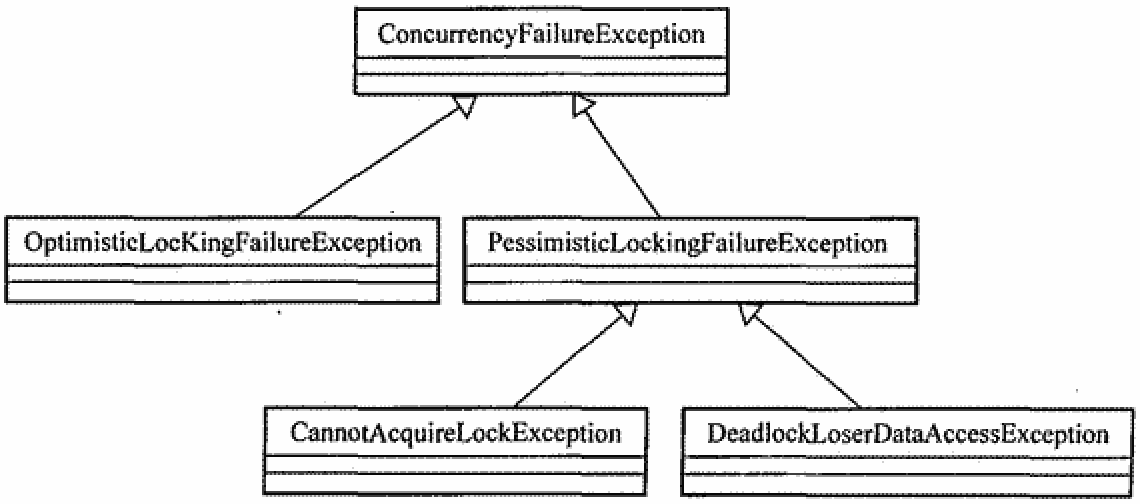

图13-2 ConcurrencyFailureException相关层次关系图

OptimisticLockingFailureException对应数据库更新时候出现乐观锁冲突的情况。PessimisticLockingFailureException自然对应的是悲观锁冲突。PessimisticLockingFailureException还可以为细分为CannotAcquireLockException和DeadlockLoserDataAccessException等特定的异常类型。

InvalidDataAccessApiUsageException。通常，该异常的出现不是因为数据资源出现了问题，而是因为我们以错误的方式，使用了特定的数据访问API。比如，使用Spring的JdbcTemplate的getForObject()方法进行查询操作，而传入的SQL查询却返回多行结果，就会抛出该异常。因为getForObject()语义上只返回一个结果对象，所以我们应该使用能够返回多行记录的查询方法，而不是只能返回单一结果的getForObject()方法。

InvalidDataAccessResourceUsageException。以错误的方式访问数据资源，就会抛出InvalidDataAccessResourceUsageException，比如，要访问数据库资源，而却传入错误的SQL，或者以其他错误方式访问数据库。各种数据访问方式会根据自身情况，抛出InvalidDataAccessResourceUsageException的子类以进一步区分详细的错误情况，比如，基于JDBC的数据访问会通过抛出org.springframework.jdbc.BadSqlGrammarException，表示访问操作传入了错误格式的SQL；而基于Hibernate的数据访问会通过抛出HibernateQueryException，表示访问操作传入的HQL语法有问题。InvalidDataAccessResourceUsageException的特定子类由相应的数据访问实现方式提供。

DataRetrievalFailureException。在要获取预期的数据却失败的时候，会抛出DataRetrievalFailureException。比如，已知某顾客存在，我们要根据该顾客号获取顾客信息却失败了的时候，可以抛出该异常。

PermissionDeniedDataAccessException。尝试访问某些数据，而自身却没有相应权限的情况下，将抛出该异常。如果某个用户没有被授予相应权限，而尝试进行权限之外的操作的时候，就会导致PermissionDeniedDataAccessException的发生。

DataIntegrityViolationException。顾名思义，数据一致性冲突异常，是在我们尝试更新数据却违反了数据一致性检查的情况下，将会抛出的异常。如数据库中已经存在主键为1的记录，我们又尝试插入同样主键记录的时候，无疑将触发该异常。通常，系统中抛出DataIntegrityViolationException表示系统的哪个部分出现了问题，这时，需要相关人员来查找到底哪个地方出现了问题。但也不排除，抛出DataIntegrityViolationException之后，可以忽略这种冲突，而将插入操作改为更新操作的情况。

假设在某个场景中有这样的系统逻辑，当向数据库中插入某条记录出现主键冲突的时候，就改为更新这条记录，要实现这样的逻辑，我们就可以捕获DataIntegrityViolationException然后进行处理（如代码清单13-8所示），而不是像通常的DataAccessException那样，在程序中不做捕获也不做处理。

代码清单13-8　捕获DataIntegrityViolationException并做相应处理的代码示例

```
try
{
   // 在当前位置执行数据访问
   // 因为主键冲突导致插入记录失败
}
catch(DataIntegrityViolationException e)
{
   logger.warn("PK Violation With....");
```

```
    // 执行更新以取代插入操作
}
```

UncategorizedDataAccessException。出现其他无法详细分类的数据访问异常，可以抛出UncategorizedDataAccessException。该异常定义为abstract。如果对于特定的数据访问方式来说，以上的异常类型无法描述当前数据访问方式中特定的异常情况，那么可以通过扩展UncategorizedDataAccessException来进一步细化特定的数据访问异常类型。

整个的Spring异常层次体系骨架结构就是这个样子，在实际运用中可以参考Spring的Javadoc获取这些异常类型和它们的子类型的更多信息。

13.5 小结

Spring提供的统一的数据访问异常层次体系，在整个Spring数据访问层中起到了提纲挈领的作用。在本章中，我们一起分析了这个数据访问异常层次体系出现的背景以及其演化过程，最后给出了针对它的详细介绍。

在各位已经对Spring的数据访问异常层次体系了如指掌之后，接下来我们将一起来了解Spring数据访问层的另一个主要特色，即JDBC API的最佳实践。

第14章 JDBC API的最佳实践

本章内容

- 基于Template的JDBC使用方式
- 基于操作对象的JDBC使用方式

Spring提供了两种使用JDBC API的最佳实践，一种是以`JdbcTemplate`为核心的基于Template的JDBC使用方式，另一种则是在`JdbcTemplate`基础之上构建的基于操作对象的JDBC使用方式。

下面让我们先从基于Template的JDBC使用方式开始看起。

14.1 基于 Template 的 JDBC 使用方式

基于Template的JDBC使用方式的最初设想和原型，需要追溯到Rod Johnson在2003年出版的*Expert One-on-One J2EE Design and Development*一书，在该书的“Practical Data Access（数据访问实践）”一章中，Rod针对JDBC使用中的一些问题提出了一套改进的实践原型，并最终将该原型完善后在Spring框架中发布。

下面是我们对这段旅程的再次回顾。

14.1.1 JDBC 的尴尬

JDBC作为Java平台访问关系数据库的标准API，其成功是有目共睹的。几乎所有Java平台的数据访问，都直接地或者间接地使用了JDBC，它是整个Java平台面向关系数据库进行数据访问的基石。

作为一个标准，无疑JDBC是成功的。但是要说JDBC在实际的使用过程中也是受人欢迎的，则不尽然了。JDBC标准主要面向较为底层的数据库操作，所以在设计过程中，比较贴近底层以提供尽可能多的功能特色。从这个角度来说，JDBC API的设计无可厚非。可是，过于贴近底层的API设计，对于开发人员的使用来说就不一定是好事了。即使执行简单的查询或者更新，开发人员也要按照API的规矩写上一大堆雷同的代码。如果不能合理封装使用JDBC API的代码，在项目中使用JDBC访问数据所出现的问题估计会让人抓狂。

对于通常的项目开发来说，如果层次划分明确，数据访问逻辑一般在相应的DAO中实现。根据功能模块的划分，可能每个开发人员都会分得或多或少的实现相应DAO的任务。假设开发人员A在分得了DAO实现任务之后开始开发，他可能开发了如代码清单14-1所示的DAO实现代码。

代码清单14-1 基于JDBC的DAO实现类示例

```
public class DAOWithA implements IDAO
{
```

```
    private final Log logger = LogFactory.getLog(DAOWithA.class);

    public int updateSomething(String sql)
    {
        int count;
        Connection con = null;
        Statement stmt = null;
        try
        {
            con = getDataSource().getConnection();
            stmt = con.createStatement();
            count = stmt.executeUpdate(sql);
            stmt.close();
            stmt = null;
        }
        catch(SQLException e)
        {
            throw new DaoException(e);
        }
        finally
        {
            if(stmt != null)
            {
                try
                {
                    stmt.close();
                }
                catch(SQLException ex)
                {
                    logger.warn("failed to close statement:"+ex);
                }
            }
            if(con != null)
            {
                try
                {
                    con.close();
                }
                catch(SQLException e)
                {
                    logger.warn("failed to close Connection:+"+ex);
                }
            }
        }
        return count;
    }
}
```

而B所负责的DAO实现中，可能也有类似的更新操作。无疑，B也要像A这样，在他的DAO中写下同样的一堆JDBC代码，类似的情况可以扩展到C、D等开发人员。如果每个开发人员都能够严格地按照JDBC编程规范进行编码还好，起码能够保证该避免的问题都能够避免掉，虽然每次都是重复基本相同的一堆代码。有一个事实是，一个团队中的开发人员是有差别的，可能有的开发人员有好的编程习惯并按照规范保证DAO实现没有问题，但无法保证其他开发人员也能够如此。很多人都或多或少地见过代码清单14-2所示的代码吧？

代码清单14-2 糟糕的JDBC API实践代码示例

```
Connection con = null;
```

14

```
    try
    {
        con  = getDataSource().getConnection();
        Statement stmt1 = con.createStatement();
        ResultSet rs1   = stmt1.executeQuery(sql);
        while(rs1.next())
        {
            String someValue = rs.getString(1);
            Statement stmt2 = con.createStatement();
            ResultSet rs2 = stmt2.executeQuery(sql2);
            while(rs1.next())
            {
                String innerValue = rs2.getString();
                ...
            }
            rs2.close();
            rs2 = null;
            stmt2.close();
            stmt2 = null;
        }
        rs1.close();
        rs1 = null;
        stmt1.close();
        stmt1 = null;
    }
    catch(SQLException e)
    {
        throw new DaoException(e);
    }
    finally
    {
        if(con != null)
        {
            try
            {
                con.close();
            }
            catch(SQLException e)
            {
                logger.warn("failed to close Connection:+"+ex);
            }
        }
    }
```

几乎程式一样的代码先不说，JDBC驱动程序是否支持嵌套的结果集也先放一边不提，这一段代码里面多个`Statement`和多个`ResultSet`的并存的情况，看着是否是曾相识呢？可能现在都从使用ORM开始，很少直接写JDBC代码了。不过，我得承认，我见过这样的代码，且对其深恶痛绝。即使你要使用多个`Statement`和多个`ResultSet`来完成一个功能，即使JDBC驱动程序也允许你这么做，但错误往往出在不经意之间，就跟上面的代码一样，你明明要对`rs2`进行遍历，却鬼使神差地写成了`rs1`。真是一个不小的“惊喜”！

这其实只是API的使用过程中的一处小插曲，当你看到应用程序中那几十甚至上百的使用JDBC实现的各个DAO实现类的时候，我可以保证你将能够发现更多的“惊喜”：

- `Statement`使用完后没关闭，而想着让`Connection`关闭的时候一并关闭，这倒是省事儿了，可是并非所有的驱动程序都有这样的行为；
- 创建了多个`ResultSet`或者`Statement`，最后却只清理了最外层的，而忽视了里层的；

- 好不容易忙活完try子句里面的数据访问逻辑，却完全忽视了使用的Connection还没有释放；

……

这时你会说，“JDBC用起来可真烦人！”嗯，不得不承认，确实如此。

不过，除了API的使用，JDBC规范在制定数据访问异常处理的时候也没能够“将革命进行到底”。

- 在最初的JDBC规范中，通过一个SQLException类型来包括一切数据访问异常情况，将SQLException声明为checked exception合适与否，我们之前已经提过，显然这是一个需要改进的地方，而最新版本的JDBC规范也就这一点进行了改进，不过这是后话；
- 除此之外，SQLException没有采用将具体的异常情况子类化，以进一步抽象不同的数据访问异常情况，而是采用ErrorCode的方式来区分数据访问过程中所出现的不同异常情况。其实，这也没什么，只要能够区分具体错误情况就行，但是，JDBC规范却把ErrorCode的规范制定留给了各个数据库提供商（当然，可能并非有意，或者有什么难言之隐）。这导致不同提供商提供的数据库对应不同的ErrorCode。进而应用程序在捕获SQLException之后，还要先看看当前使用的是什么数据库，然后再从SQLException中通过getErrorCode()取得相应ErrorCode，并与数据库提供商提供的ErrorCode列表进行对比，最终才能搞清楚到底哪里出了问题。如果当初JDBC规范能够明确规定“ErrorCode=1代表数据库连接不上，ErrorCode=2代表要访问的表不存在……”现在处理SQLException的时候也不会如此痛苦了。

注意 Java平台从1.1版本开始引入JDBC，迄今已经经历了几个大的版本变动，JDBC标准的制定也吸取了Spring框架的JDBC抽象层的部分经验，比如对数据访问异常的层次体系的处理方面。我们可以在最新的JDBC规范中看到这些改进后的亮点。

针对JDBC API在使用中容易出错，使用烦琐的问题，以及SQLException对数据访问异常处理能力不足尚待改进等情况，Spring框架提出了一套针对JDBC使用方面的框架类，以改进JDBC API使用过程中的种种不便甚至不合理之处，帮助我们进一步提高开发过程中使用JDBC进行数据访问的开发效率。

14.1.2 JdbcTemplate 的诞生

为了解决JDBC API在实际使用中的种种尴尬局面，Spring框架提出了org.springframework.jdbc.core.JdbcTemplate作为数据访问的Helper类。JdbcTemplate是整个Spring数据抽象层提供的所有JDBC API最佳实践的基础，框架内其他更加方便的Helper类以及更高层次的抽象，全部构建于JdbcTemplate之上。抓住JdbcTemplate，就抓住了Spring框架JDBC API最佳实践的核心。

概括地说，JdbcTemplate主要关注如下两个事情[①]。

- 封装所有基于JDBC的数据访问代码，以统一的格式和规范来使用JDBC API。所有基于JDBC的数据访问需求现在全部通过JdbcTemplate进行，从而避免了让烦琐易错的基于JDBC API的数据访问代码散落于系统各处。
- 对SQLException所提供的异常信息在框架内进行统一转译，将基于JDBC的数据访问异常纳入Spring自身的异常层次体系中，统一了数据接口的定义，简化了客户端代码对数据访问异常的处理。

① 我们这里主要是从JdbcTemplate简化JDBC使用的角度来看待它所关注的事情。稍后你将看到，实际上，为了集成Spring的事务抽象层，JdbcTemplate在实现的时候还需要关注更多事情。

因为JdbcTemplate主要是通过模板方法模式对基于JDBC的数据访问代码进行统一封装，所以，在详细了解JdbcTemplate的实现之前，我们有必要先简单回顾一下模板方法模式。

1. 模板方法模式简介

模板方法模式（Template Method Pattern）主要用于对算法或者行为逻辑进行封装，即如果多个类中存在某些相似的算法逻辑或者行为逻辑，可以将这些相似的逻辑提取到模板方法类中实现，然后让相应的子类根据需要实现某些自定义的逻辑。

举个例子来说，所有汽车，不管是宝马还是夏利，它们的驾驶流程基本是固定的。实际上，除了少数细节不同，大部分的流程是一样的，基本上是如下所示的流程说明：

(1) 点火启动汽车；

(2) 踩刹车，挂前进挡位（手动挡和自动挡这一步存在差异）；

(3) 放下手动制动器（俗称手刹）；

(4) 踩油门启动车辆运行。

那么，我们可以声明一个模板方法类，将确定的行为以模板的形式定义，而将不同的行为留给相应的子类实现。我们的模板类定义如代码清单14-3所示。

代码清单14-3　抽象模板类Vehicle的定义

```
public abstract Vehicle
{
    public final void drive()
    {
        startTheEngine(); // 点火启动汽车
        putIntoGear(); // 踩刹车，挂前进挡位
        looseHandBrake(); // 放下手动制动器
        stepOnTheGasAndGo(); // 踩油门启动车辆运行
    }

    protected abstract void putIntoGear();

    private void stepOnTheGasAndGo() {
        // ...
    }

    private void looseHandBrake() {
        // ...
    }

    private void startTheEngine() {
        // ...
    }
}
```

drive()方法就是我们声明的模板方法，它声明为final，也就是说，方法内的逻辑是不可变更的。车辆的入档因自动档车辆和手动挡车辆而有所不同，所以将putIntoGear()声明为抽象方法留给相应的具体子类实现，如代码清单14-4中的子类定义。

代码清单14-4　模板类Vehicle对应的子类定义示例

```
// 自动挡汽车
public class VehicleAT extends Vehicle
{
    @Override
```

```
    protected void putIntoGear() {
        // 挂前进挡位
    }
}
// 手动挡汽车
public class VehicleMT extends Vehicle
{
    @Override
    protected void putIntoGear() {
        // 踩离合器
        // 挂前进挡位
    }
}
```

这样一来，就不需要每个子类中都声明并实现共有的逻辑，而只需要每个子类实现特有的逻辑就行了。

2. JdbcTemplate的演化

如果我们回头看一下最初的直接使用JDBC API进行数据访问的代码，就会发现，不管这些代码是由谁负责的，也不管这些代码所实现的数据访问逻辑如何，除了小部分的差异之外，所有这些代码几乎都是按照同样的一套流程下来的，如下。

(1) `con = getDataSource().getConnection();` 取得数据库连接。

(2) `stmt = con.createStatement();`或者`ps = con.prepareStatement(..);`根据Connection创建相应的`Statement`或者`PreparedStatement`。

(3) `stmt.executeUpdate(sql);` 或者 `ps.executeUpdate(..);` 根据传入的SQL语句或者参数，借助`Statement`或者`PreparedStatement`进行数据库的更新或者查询。如果是查询操作，那么可以按如下代码对结果集进行遍历，抽取查询后的结果：

```
ResultSet rs = stmt.executeQuery(sql);
while(rs.next())
{
    processResultRow(rs);
}
```

(4) `stmt.close(); stmt = null;` 关闭相应的`Statement`或者`PreparedStatement`。

(5) `catch(SQLException e){...}`处理相应的数据库访问异常。

(6) `finally{con.close();}` 关闭数据库连接以避免连接泄漏导致系统崩溃。

对于多个DAO中充斥的几乎相同的JDBC API的使用代码，我们也可以采用模板方法模式，对这些基于JDBC API的数据访问代码进行重构，杜绝因个人使用不当所导致的种种问题。我们所要做的，只是将共有的一些行为提取到模板方法类中，而特有的操作，比如每次执行不同的更新，或者每次针对不同的查询结果进行不同的处理等行为，则放入具体子类中。这样，我们就有了一个`JdbcTemplate`的雏形（如代码清单14-5所示）。

14

代码清单14-5 JdbcTemplate雏形代码示例

```
public abstract class JdbcTemplate
{
public final Object execute(String sql)
{
    Connection con = null;
    Statement stmt = null;
    try
```

```
        {
            con = getConnection();
            stmt = con.createStatement();
            Object retValue = executeWithStatement(stmt,sql);
            return retValue;
        }
        catch(SQLException e)
        {
            DataAccessException ex = translateSQLException(e);
            throw ex;
        }
        finally
        {
            closeStatement(stmt);
            releaseConnection(con);
        }
    }

    protected abstract Object executeWithStatement(Statement stmt,String sql);

    ...// 其他方法定义
    }
```

这样处理之后，JDBC代码的使用得到了规范（进行数据访问的时候，每次使用的JDBC代码都几乎相同），异常处理和连接释放等问题也得到了统一的管理。但是，只使用模板方法模式还不足以提供方便的数据访问Helper类。顶着abstract帽子的JdbcTemplate作为Helper类，不能够独立使用暂且不说，让我们每次进行数据访问的时候都要给出一个相应的子类实现，这也实在太不地道了。

所以，Spring框架在实现JdbcTemplate的时候，除了使用模板方法模式之外，还引入了相应的Callback接口定义，以避免每次使用该Helper类的时候都需要进行子类化。当引入称为Statement-Callback的接口定义之后（如下所示）：

```
public interface StatementCallback{
    Object doWithStatement(Statement stmt);
}
```

我们的JdbcTemplate就可以摆脱abstract的帽子，作为一个真正的Helper类而独立存在了（见代码清单14-6）。

代码清单14-6　JdbcTemplate原型代码定义

```
public class JdbcTemplate
{
public final Object execute(StatementCallback callback)
{
    Connection con = null;
    Statement stmt = null;
    try
    {
        con = getConnection();
        stmt = con.createStatement();

        Object retValue = callback.doWithStatement(stmt);
        return retValue;
    }
    catch(SQLException e)
    {
        DataAccessException ex = translateSQLException(e);
```

```
            throw ex;
        }
        finally
        {
            closeStatement(stmt);
            releaseConnection(con);
        }
    }
    ...// 其他方法定义
    }
```

要在相应的DAO实现类中使用JdbcTemplate，只需要根据情况提供参数和相应的Statement-Callback就行了，如下代码演示了JdbcTemplate原型的使用：

```
JdbcTemplate jdbcTemplate = ...;
final String sql = "update ...";
StatementCallback callback = new StatementCallback(){
        public Obejct doWithStatement(Statement stmt)
        {
            return new Integer(stmt.executeUpdate(sql));
        }
    };
jdbcTemplate.execute(callback);
```

现在，开发人员只需要关心与数据访问逻辑相关的东西，对于JDBC底层相关的细节则不用过多地考虑。最主要的是，令人恼火的数据库连接没释放的问题，将再也不会来烦我们了。

> **注意** Callback接口与模板方法类之间的关系可以看作是服务与被服务的关系，模板方法类想要Callback做事，就要提供相应的资源（在这里是通过doWithStatement方法公开Statement给Callback接口）。Callback使用提供的资源做事，完事之后，模板方法类来处理公开的资源，Callback接口不需要关心这些。
>
> 实际上，Java中没有Ruby语言类似的Block的语法结构，所以，我们只好使用匿名内部类来完成同样的功能。幸运的是，匿名内部类也不算难用。现在对于要不要在Java中引入Closure语法结构的讨论也在进行中，据说Java7中将会引入对Closure的支持，那时或许就可以直接使用Closure语法来代替匿名内部类了。

到此为止，我们说的只是JdbcTemplate实现的中心思想。实际上，JdbcTemplate在实现的细节上要考虑许多的东西，所以，还是来看一下Spring中的JdbcTemplate到底是一个什么样的实现结构吧！

org.springframework.jdbc.core.JdbcTemplate的继承层次比较简单，如图14-1所示。

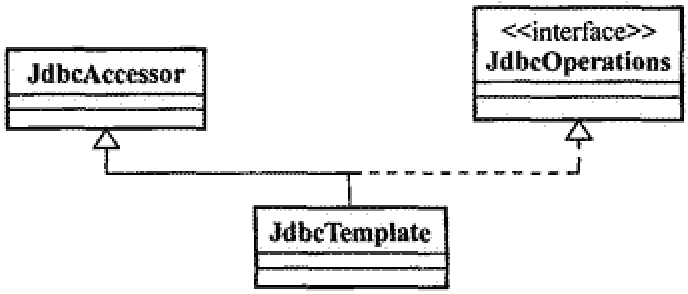

图14-1 JdbcTemplate继承层次

org.springframework.jdbc.core.JdbcOperations接口定义界定了JdbcTemplate可以使用

的JDBC操作集合，该接口提供的操作声明，从查询到更新无所不包，详情不在这里罗列了，可以查询该接口定义的Javadoc了解其中定义的所有可用的JDBC操作。JdbcTemplate的直接父类是org.springframework.jdbc.support.JdbcAccessor，这是一个抽象类，主要为子类提供一些公用的属性，如下。

- **DataSource**。javax.sql.DataSource是JDBC 2.0之后引入的接口定义，用来替代基于java.sql.DriverManager的数据库连接创建方式。DataSource的角色可以看作JDBC的连接工厂（ConnectionFactory），具体实现可以引入对数据库连接的缓冲池以及分布式事务支持。所以，基本上javax.sql.DataSource现在应该作为获取数据库资源的统一接口。
  Spring数据访问层对数据库资源的访问，全部建立在javax.sql.DataSource标准接口之上，通过超类JdbcAccessor，JdbcTemplate自然也是以此为基准。
- **SQLExceptionTranslator**。Spring将对SQLException的转译这一工作，抽象为特定的接口，也就是org.springframework.jdbc.support.SQLExceptionTranslator。我们将在稍后，对Spring框架中如何实现SQLException到其统一的数据访问异常体系的转译做详细介绍，现在只需要明白，通过超类JdbcAccessor定义的有关设置或者获取SQLExceptionTranslator的方法，JdbcTemplate可以在处理SQLException的时候，委托具体的SQLExceptionTranslator实现类来进行SQLException的转译。

这样，在JdbcAccessor的支持下，JdbcTemplate就开始为了实现JdbcOperations所规定的目标而努力奋斗了！

JdbcTemplate中各种模板方法按照其通过相应Callback接口所公开的API自由度的大小，可以简单划分为如下4组。

- **面向Connection的模板方法**。属于这一组的模板方法通过org.springframework.jdbc.core.ConnectionCallback回调接口所公开的java.sql.Connection进行数据访问。虽然关于Connection的获取和释放不需要关心，但通过ConnectionCallback所公开的API使用自由度还是很大，所以，除非特殊情况，比如，集成遗留系统的数据访问，通常情况下，应避免直接使用面向Connection层面的模板方法进行数据访问。
- **面向Statement的模板方法**。面向Statement的模板方法主要处理基于静态的SQL的数据访问请求。该组模板方法通过org.springframework.jdbc.core. StatementCallback回调接口对外公开java.sql.Statement类型的操作句柄。该方式缩小了回调接口内的权限范围，但是提高了API使用上的安全性和便捷性。
- **面向PreparedStatement的模板方法**。对于使用包含查询参数的SQL请求来说，使用PreparedStatement可以让我们免于SQL注入的攻击。而在使用PreparedStatement之前，需要根据传入的包含参数的SQL对其进行创建，所以，面向PreparedStatement的模板方法会通过org.springframework.jdbc.core.PreparedStatementCreator回调接口公开Connection以允许PreparedStatement的创建。另外，PreparedStatement创建之后，会公开给org.springframework.jdbc.core.PreparedStatementCallback回调接口，以支持其使用PreparedStatement进行数据访问。
- **面向CallableStatement的模板方法**。JDBC支持使用CallalbeStatement进行数据库存储过程的访问。面向CallableStatement的模板方法会通过org.springframework.jdbc.core.CallableStatementCreator公开相应的Connection以便创建用于调用存储过程的CallableStatement。之后，再通过org.springframework.jdbc.core.CallableStat-

ementCallback公开创建的CallableStatement操作句柄，实现基于存储过程的数据访问。

每一组中的模板方法都有一个核心的方法实现，其他属于同一组的重载的模板方法，会调用这个核心的方法实现来完成最终的工作。以面向Statement的模板方法分组为例，使用Statement-Callback回调接口作为方法参数的execute()方法是这一组模板方法的核心实现（见代码清单14-7）。

代码清单14-7 面向Statement的核心模板方法定义代码摘录

```
// --- 摘自Spring框架JdbcTemplate源码 ---
public Object execute(StatementCallback action) throws DataAccessException {
Assert.notNull(action, "Callback object must not be null");

Connection con = DataSourceUtils.getConnection(getDataSource());
Statement stmt = null;
try {
    Connection conToUse = con;
    if (this.nativeJdbcExtractor != null && ➥
    this.nativeJdbcExtractor.isNativeConnectionNecessaryForNativeStatements()) {
        conToUse = this.nativeJdbcExtractor.getNativeConnection(con);
        }
    stmt = conToUse.createStatement();
    applyStatementSettings(stmt);
    Statement stmtToUse = stmt;
    if (this.nativeJdbcExtractor != null) {
        stmtToUse = this.nativeJdbcExtractor.getNativeStatement(stmt);
    }
    Object result = action.doInStatement(stmtToUse);
    handleWarnings(stmt.getWarnings());
    return result;
}
catch (SQLException ex) {
    // Release Connection early, to avoid potential connection pool deadlock
    // in the case when the exception translator hasn't been initialized yet.
    JdbcUtils.closeStatement(stmt);
    stmt = null;
    DataSourceUtils.releaseConnection(con, getDataSource());
    con = null;
    throw getExceptionTranslator().translate("StatementCallback", getSql(action), ex);
}
finally {
    JdbcUtils.closeStatement(stmt);
    DataSourceUtils.releaseConnection(con, getDataSource());
}
```

其他模板方法会根据自身的方法签名，构建相应的StatementCallback实例以调用execute(StatementCallback)方法，如代码清单14-8所示。

代码清单14-8 调用execute(StatementCallback)的模板方法定义示例

```
// --- 摘自JdbcTemplate源代码 ---
public void execute(final String sql) throws DataAccessException {
if (logger.isDebugEnabled()) {
    logger.debug("Executing SQL statement [" + sql + "]");
}

class ExecuteStatementCallback implements StatementCallback, SqlProvider {
    public Object doInStatement(Statement stmt) throws SQLException {
        stmt.execute(sql);
        return null;
```

```
        }
        public String getSql() {
            return sql;
        }
    }
    execute(new ExecuteStatementCallback());
    }
```

同一组内的模板方法，可以根据使用的方便性进行增加，只要在实现的时候，将相应条件以对应该组的回调接口进行封装，最终调用当前组的核心模板方法即可。

3. 使用DataSourceUtils进行Connection的管理

如果你稍微关注JdbcTemplate的实现代码，就会发现JdbcTemplate在取得具体的数据库Connection的时候，不是从相应的DataSource中通过getConnection()方法直接取得可用的Connection对象，如以下代码所示：

```
Connection con = dataSource.getConnection();
```

而是使用了一个DataSourceUtils工具类从指定的DataSource中取得相应的Connection，如以下代码所示：

```
Connection con = DataSourceUtils.getConnection(getDataSource());
```

这是为什么呢？

实际上，如果我们要实现一个功能单一的JdbcTemplate的话，通过DataSource来getConnection()是完全可以的。只不过，Spring所提供的JdbcTemplate要关注更多的东西，所以，在从DataSource中取得连接的时候需要多做一点儿事情而已。

org.springframework.jdbc.datasource.DataSourceUtils提供相应的方法，用来从指定的DataSource获取或者释放连接。与直接从DataSource取得Connection不同，DataSourceUtils会将取得的Connection绑定到当前线程，以便在使用Spring提供的统一事务抽象层进行事务管理的时候使用。有关Spring中统一的事务抽象概念将在第15章中详细阐述，现在只需要知道，使用DataSourceUtils作为Helper类从DataSource中取得Connection的方式，基本上比直接从DataSource中取得Connection的方式就多了这些东西。

4. 使用NativeJdbcExtractor来获得"真相"

对于DataSource实现来说，特别是J2EE应用服务器所提供的DataSource实现，出于事务管理或者其他有关资源管理的目的，当从这些DataSource实现中请求相应的Connection以及相关的Statement的时候，它们会返回对应最初Connection以及Statement对象的代理对象。

这么一处理，我们就得忽略各种Connection的特异性，而只能以java.sql.Connection接口定义的方式使用它。可是，如果我们要获得数据库驱动程序提供的原始Connection实现类（例如，oracle.jdbc.OracleConnection），以便使用特定于数据库的特色功能的话，使用DataSource所返回的代理类将无法做到。

所以，这也就是为什么JdbcTemplate在使用具体的Connection或者Statement之前，会首先检查是否需要使用驱动程序提供的具体实现类，而不是相应的代理对象（如以下代码所示）：

```
if (this.nativeJdbcExtractor != null) {
    // Extract native JDBC Connection, castable to OracleConnection or the like.
    conToUse = this.nativeJdbcExtractor.getNativeConnection(con);
}
...
```

```
    if (this.nativeJdbcExtractor != null) {
        stmtToUse = this.nativeJdbcExtractor.getNativeStatement(stmt);
    }
    ...
```

JdbcTemplate内部定义了一个NativeJdbcExtractor类型的实例变量，当我们想要使用数据库驱动所提供的原始API的时候，可以通过JdbcTemplate的setNativeJdbcExtractor (NativeJdbcExtractor)方法设置相应的NativeJdbcExtractor实现类。这样，设置后的NativeJdbcExtractor实现类将负责剥离相应的代理对象，取得真正的目标对象供我们使用。

Spring默认提供面向Commons DBCP、C3P0、Weblogic、WebSphere等数据源的NativeJdbcExtractor实现类。

- CommonsDbcpNativeJdbcExtractor。为Jakarta Commons DBCP数据库连接池所提供的NativeJdbcExtractor实现类。
- C3P0NativeJdbcExtractor。为C3P0数据库连接池所提供的NativeJdbcExtractor实现类。
- WebLogicNativeJdbcExtractor。为Weblogic所准备的NativeJdbcExtractor实现类。
- WebSphereNativeJdbcExtractor。为WebSphere所准备的NativeJdbcExtractor实现类。

可以从org.springframework.jdbc.support.nativejdbc包获得所有这些实现类的更多信息。

5. 控制JdbcTemplate的行为

JdbcTemplate在通过Statement或者PreparedStatement等进行具体的数据操作之前，会调用如下所示的代码：

```
applyStatementSettings(stmt);
或者
applyStatementSettings(ps);
或者
applyStatementSettings(cs);
```

这行代码有什么用处呢？

实际上，从代码清单14-9中可以看出，通过该方法，我们可以控制查询的一些行为，比如控制每次取得的最大结果集，以及查询的超时时间（timeout）等。

代码清单14-9 applyStatementSettings方法定义代码摘录

```
protected void applyStatementSettings(Statement stmt) throws SQLException {
    int fetchSize = getFetchSize();
    if (fetchSize > 0) {
        stmt.setFetchSize(fetchSize);
    }
    int maxRows = getMaxRows();
    if (maxRows > 0) {
        stmt.setMaxRows(maxRows);
    }
    DataSourceUtils.applyTimeout(stmt, getDataSource(), getQueryTimeout());
}
```

可以通过相应的setter()方法对JdbcTemplate声明的fetchSize、maxRows以及queryTimeout属性进行设置。比如，如果某个查询可能返回的结果集很大，一次抽取的话可能导致程序内存溢出（OutOfMemory），那么我们就可以通过设置fetchSize来指定每次最多抽取1 000行，如下所示：

```
JdbcTemplate jt = new JdbcTemplate(..);
jt.setFetchSize(1000);
// 使用jt进行数据访问
```

很简单，不是吗？

6. **SQLException到DataAccessException体系的转译**

因为JdbcTemplate直接操作JDBC API，所以它需要捕获在此期间可能发生的SQLException并进行处理，处理的宗旨是将SQLException转译到Spring的数据访问异常层次体系，以统一数据访问异常的处理方式。

JdbcTemplate将SQLException转译到Spring数据访问异常层次体系这部分工作转交给了org.springframework.jdbc.support.SQLExceptionTranslator接口来完成，该接口的定义如下：

```
public interface SQLExceptionTranslator {
    DataAccessException translate(String task, String sql, SQLException ex);
}
```

该接口有两个主要实现类，分别为org.springframework.jdbc.support.SQLErrorCodeSQLExceptionTranslator和org.springframework.jdbc.support.SQLStateSQLExceptionTranslator，如图14-2所示。

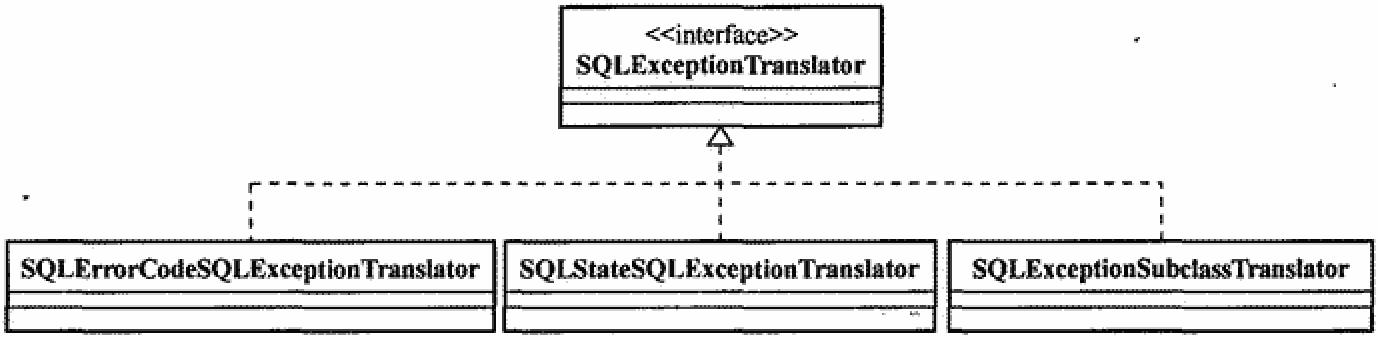

图14-2　SQLExceptionTranslator体系

图14-2中的SQLExceptionSubclassTranslator是Spring 2.5新增加的SQLExceptionTranslator实现类，主要用于将随JDK 6发布的JDBC 4版本中新定义的异常体系转化到Spring的数据访问异常体系。对于之前的版本，该实现类自然是用不上了。

SQLErrorCodeSQLExceptionTranslator会基于SQLException所返回的ErrorCode进行异常转译。通常情况下，根据各数据库提供商所提供的ErrorCode进行分析要比基于SQLState的方式准确得多。默认情况下，JdbcTemplate采用SQLErrorCodeSQLExceptionTranslator进行SQLException的转译工作。只有当通过ErrorCode方式无法提供足够信息的时候，才会转而求助于SQLStateSQLExceptionTranslator。

SQLStateSQLExceptionTranslator根据SQLException.getSQLState()所返回的信息进行异常转译，虽然基于SQLState的方式理论上应该是最通用的（因为SQLState要求符合XOPEN SQLstate规范或者SQL 99规范），但各数据库厂商在执行上存在差异。所以，更多时候，我们偏向于基于ErrorCode的方式。

如果JdbcTemplate默认的SQLErrorCodeSQLExceptionTranslator无法满足当前异常转译的需要，我们可以扩展SQLErrorCodeSQLExceptionTranslator，使它支持更多的情况，这有两种办法，分别是提供SQLErrorCodeSQLExceptionTranslator的子类或者在Classpath的根路径下添加固定的配置文件。当然，在此之前，我们还是先搞清楚为什么要这么做吧。

SQLErrorCodeSQLExceptionTranslator进行异常转译的流程大体上如以下所述。

(1) SQLErrorCodeSQLExceptionTranslator中定义了如下的自定义异常转译方法：

```
DataAccessException customTranslate(String task, String sql, SQLException sqlEx)
```

程序流程首先会检查该自定义异常转译方法是否能够对当前传入的SQLException进行转译，如果可以，则直接返回转译后的DataAccessException类型；如果该方法返回null，则表示当前方法无法对传入的SQLException进行转译，程序流程将进入下一步，寻求其他方式进行转译。在SQLErrorCodeSQLExceptionTranslator中，该方法直接返回null，所以，程序流程进入下一步。

(2) 如果应用程序运行于Java 6之上，那么，SQLErrorCodeSQLExceptionTranslator会尝试让SQLExceptionSubclassTranslator进行异常的转译。对于使用Java 6之前版本的应用来说，这一步基本可以忽略，了解即可。

(3) 使用org.springframework.jdbc.support.SQLErrorCodesFactory所加载的SQLErrorCodes进行异常转译，其中，SQLErrorCodesFactory加载SQLErrorCodes的流程为：

- 加载位于Spring发布jar包中org/springframework/jdbc/support/sql-error-codes.xml路径下的记载了各个数据库提供商ErrorCode的配置文件，提取相应的SQLErrorCodes。
- 如果发现当前应用的Classpath的根路径下存在名称为sql-error-codes.xml的配置文件，则加载该文件内容，并覆盖默认的ErrorCode定义。

(4) 如果基于ErrorCode的异常转译还是搞不定的话，SQLErrorCodeSQLExceptionTranslator将求助于SQLStateSQLExceptionTranslator，最后使用基于SQLState的方式进行SQLException到Spring数据访问异常体系的转译工作。

在以上SQLErrorCodeSQLExceptionTranslator进行异常转译的整个流程中，我们可以在如下两个点插入自定义的异常转译逻辑。

(a) 流程的第一步首先检查customTranslate方法是否可以完成转译工作，SQLErrorCodeSQLExceptionTranslator默认实现返回null，不做自定义的转译工作。所以，可以继承SQLErrorCodeSQLExceptionTranslator以实现它的一个子类，在子类中覆写该方法，然后使用这个子类替代SQLErrorCodeSQLExceptionTranslator。

(b) SQLErrorCodesFactory在为SQLErrorCodeSQLExceptionTranslator加载ErrorCode配置文件的时候，尝试从Classpath的根路径下加载自定义的名称为sql-error-codes.xml的配置文件以覆盖默认的定义，我们可以提供一个名称为sql-error-codes.xml的配置文件，在其中增加自定义的ErrorCode定义。

下面是使用这两种方式进行自定义的异常转译的具体实施情况。

- **扩展SQLErrorCodeSQLExceptionTranslator**

扩展SQLErrorCodeSQLExceptionTranslator以做到自定义异常转译，虽然是最直接有效的方式，却算不上方便。

首先，需要定义SQLErrorCodeSQLExceptionTranslator的子类，然后如代码清单14-10所示覆写它的customTranslate方法。

代码清单14-10 自定义SQLErrorCodeSQLExceptionTranslator示例

```
public class ToySqlExceptionTranslator extends SQLErrorCodeSQLExceptionTranslator {
    ...
    @Override
    protected DataAccessException customTranslate(String task, String sql,➥
    SQLException sqlEx) {
```

```
        if(sqlEx.getErrorCode() == 123456)
        {
          String msg = new StringBuffer()
            .append("unexpected data access exception raised when executing ")
            .append(task)
            .append(" with SQL>")
            .append(sql)
            .toString();
            return new UnexpectedDataAccessException(msg,sqlEx);
        }
        return null;
        }
}
```

在这里，假设当数据库返回的错误码为123456的时候，将抛出UnexpectedDataAccessException类型的异常（或者其他自定义的DataAccessException实现），除此之外，返回null以保证其他的异常转译依然采用超类SQLErrorCodeSQLExceptionTranslator原来的逻辑进行。

为了能够让自定义的异常转译逻辑生效，我们需要让JdbcTemplate使用我们的ToySqlExceptionTranslator，而不是默认的SQLErrorCodeSQLExceptionTranslator，所以，需要如以下代码所示，将自定义的ToySqlExceptionTranslator设置给相应的JdbcTemplate使用：

```
DataSource dataSource = ...;
JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

// 设置自定义的SQLErrorCodeSQLExceptionTranslator
SQLErrorCodeSQLExceptionTranslator sqlExTranslator = new ToySqlExceptionTranslator();
sqlExTranslator.setDataSource(dataSource);
jdbcTemplate.setExceptionTranslator(sqlExTranslator);
// 现在可以使用jdbcTemplate进行数据访问了
```

至此，扩展SQLErrorCodeSQLExceptionTranslator已经达到了自定义异常转译的目的。不过，与下面的方式比起来，具体实践上面则要稍逊一筹。

- 提供sql-error-codes.xml自定义配置

在Classpath的根路径下放置名称为sql-error-codes.xml的配置文件，格式需要与默认的org/springframework/jdbc/support/sql-error-codes.xml文件格式相同。

org/springframework/jdbc/support/sql-error-codes.xml的内容摘录如代码清单14-11所示。

代码清单14-11　org/springframework/jdbc/support/sql-error-codes.xml内容摘录

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN" "http://www.springframework.org/➥
dtd/spring-beans-2.0.dtd">

<!--
    - Default SQL error codes for well-known databases.
    - Can be overridden by definitions in a "sql-error-codes.xml" file
    - in the root of the class path.
    -
    - If the Database Product Name contains characters that are invalid
    - to use in the id attribute (like a space) then we need to add a property
    - named "databaseProductName"/"databaseProductNames" that holds this value.
    - If this property is present, then it will be used instead of the id for
    - looking up the error codes based on the current database.
    -->
```

```
<beans>
    <bean id="DB2" class="org.springframework.jdbc.support.SQLErrorCodes">
        <property name="databaseProductName">
            <value>DB2*</value>
        </property>
        <property name="badSqlGrammarCodes">
            <value>-204,-206,-301,-408</value>
        </property>
        <property name="dataAccessResourceFailureCodes">
            <value>-904</value>
        </property>
        <property name="dataIntegrityViolationCodes">
            <value>-803</value>
        </property>
        <property name="deadlockLoserCodes">
            <value>-911,-913</value>
        </property>
    </bean>

    ...

</beans>
```

实际上，它就是一个基本的基于DTD的Spring IoC容器配置文件，只不过，配置的class类型是固定的。

该配置文件针对每个数据库类型都提供了一个`org.springframework.jdbc.support.SQLErrorCodes`类型的bean定义，然后根据各个数据库的情况，通过相应的setter方法为`SQLErrorCodes`设置合适的ErrorCode或者数据库产品别名等。

要扩展异常转译，我们可以根据情况提供Classpath根路径下的`sql-error-codes.xml`配置文件内容。

- 如果默认的`org/springframework/jdbc/support/sql-error-codes.xml`配置文件中，缺少对应当前应用程序所使用的数据库的bean定义配置，那么直接复制现有的某个bean定义到Classpath根路径下的`sql-error-codes.xml`中，然后修改相应属性值即可。如代码清单14-12添加了一个针对虚构的MyDB的`SQLErrorCodes`类型bean定义。

代码清单14-12　虚构的MyDB对应的`SQLErrorCodes`类型bean定义

```
<bean id="MyDB" class="org.springframework.jdbc.support.SQLErrorCodes">
    <property name="databaseProductName">
        <value>MyDBAlias</value>
    </property>
    <property name="badSqlGrammarCodes">
        <value>000</value>
    </property>
    <property name="dataAccessResourceFailureCodes">
        <value>111</value>
    </property>
    <property name="dataIntegrityViolationCodes">
        <value>222</value>
    </property>
    <property name="deadlockLoserCodes">
        <value>333</value>
    </property>
</bean>
```

注意，我们把数据库名称改为了MyDB之类，相应的错误号也做了改变。当然，这毕竟是例子，

所有的东西都是伪造的，如果真要提供这样的配置内容，需要根据具体的数据库信息才可进行。

- 如果需要扩展的自定义异常在转译的时候是针对现有的数据库，那么，我们就从org/springframework/jdbc/support/sql-error-codes.xml中，复制对应应用程序使用的数据库的bean定义内容，到Classpath根路径下的sql-error-codes.xml中。只不过，现在我们要增加新的元素，如代码清单14-13所示。

代码清单14-13 为DB2数据的异常转译增加自定义转译逻辑配置代码示例

```
<bean id="DB2" class="org.springframework.jdbc.support.SQLErrorCodes">
    <property name="databaseProductName">
        <value>DB2*</value>
    </property>
    <property name="customTranslations">
      <list>
          <bean class="org.springframework.jdbc.support.CustomSQLErrorCodesTranslation">
            <property name="errorCodes">
                  <value>123456</value>
            </property>
            <property name="exceptionClass">
                <value>...ToySqlExceptionTranslator</value>
            </property>
          </bean>
      </list>
    </property>
</bean>
```

通过SQLErrorCodes的customTranslations属性传入要扩展的自定义异常转译的必要信息即可。customTranslations是org.springframework.jdbc.support.CustomSQLErrorCodesTranslation类型数组，我们可以通过该属性指定多个CustomSQLErrorCodesTranslation实例。而CustomSQLErrorCodesTranslation可以简单看作ErrorCode到异常类型的映射包装类，仅此而已。

基本上，使用Classpath根路径下的sql-error-codes.xml，就是简单的配置，便可以完成自定义异常转译的扩展工作。

14.1.3 JdbcTemplate 和它的兄弟们

我不得不承认的一点就是，之前对JdbcTemplate的实现原理和细节说了那么多，无非是想让读者知道，当需要一个类似的轮子的时候，我们该如何去造一个出来。但是，当有现成的轮子存在的时候，我希望我们去使用这个现成的轮子，而不是耗费人力物力去重造一个。所以，下面主要是告诉读者有哪些轮子可用，这些轮子又是如何使用的。

最初，Spring框架只提供了JdbcTemplate这一个实现。但随着Java版本升级，并且考虑到使用中的便利性等问题，Spring在新发布的版本中又为JdbcTemplate添加了两位兄弟，一个是org.springframework.jdbc.core.simple.SimpleJdbcTemplate，主要面向Java 5提供的一些便利；另一个是org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate，可以在SQL中使用名称代替原先使用的?占位符。下面，让我们先从JdbcTemplate开始，亲身领略一下Spring所提供的数据访问方式的便利和优雅。

1. 使用JdbcTemplate进行数据访问

- *初始化*JdbcTemplate

JdbcTemplate的初始化很容易，只要通过构造方法传入它所使用的DataSource就可以。如果使

用Jakarta Commons DBCP，那么，初始化代码如代码清单14-14所示。

代码清单14-14 使用Commons DBCP创建JdbcTemplate代码示例

```
BasicDataSource dataSource = new BasicDataSource();
dataSource.setDriverClassName("com.mysql.jdbc.Driver");
dataSource.setUrl("jdbc:mysql://localhost/mysql?useUnicode=true&amp;characterEncoding= ➥
utf8&amp;failOverReadOnly=false&amp;roundRobinLoadBalance=true");
dataSource.setUsername("user");
dataSource.setPassword("password");

JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
// 现在可以使用jdbcTemplate进行数据访问了
```

我们也可以通过无参数的构造方法来实例化JdbcTemplate，然后通过setDataSource()来设置所使用的DataSource。

当然，这仅限于使用编码的方式来初始化JdbcTemplate。如果应用程序使用Spring的IoC容器，那么JdbcTemplate的初始化工作就可以转移到容器的配置文件中（见代码清单14-15）。

代码清单14-15 Spring的IoC容器中的JdbcTemplate使用配置示例

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method ➥
="close">
    <property name="url">
        <value>${jdbc.url}</value>
    </property>
    <property name="driverClassName">
        <value>${jdbc.driver}</value>
    </property>
    <property name="username">
        <value>${jdbc.username}</value>
    </property>
    <property name="password">
        <value>${jdbc.password}</value>
    </property>
    ...
</bean>

<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
    <property name="dataSource">
        <ref bean="dataSource"/>
    </property>
</bean>
```

之后，想把JdbcTemplate注入哪个依赖于它的对象都可以。不过，这里需要注意的不是JdbcTemplate的配置，而是DataSource的配置，我们使用了自定义的销毁对象的回调方法(destroy-method="close")，以确保应用退出后，数据库连接可以关闭。

好了，JdbcTemplate初始化完成之后，我们就可以大展拳脚了。

- *基于JdbcTemplate的数据访问*

这里仅从整体上对如何使用JdbcTemplate进行基本的数据访问进行介绍，更多JdbcTemplate使用的细节请参照Spring参考文档以及相应的Javadoc文档。

使用JdbcTemplate查询数据

JdbcTemplate针对数据查询提供了多个重载的模板方法，我们可以根据需要选用不同的模板方

14

法。如果查询很简单，仅是传入相应SQL或者相关参数，然后取得一个结果，那么我们可以选择如下一组便利的模板方法：

- int queryForInt(String sql)
- int queryForInt(String sql, Object[] args)
- long queryForLong(String sql)
- long queryForLong(String sql, Object[] args)
- Object queryForObject(String sql, Class requiredType)
- Object queryForObject(String sql, Object[] args, Class requiredType)
- Map queryForMap(String sql)
- Map queryForMap(String sql, Object[] args)

如果所查询的结果就包含一列数字型的结果，或者使用了SQL函数，或者其他单列的结果，那么就可以直接通过这组便利的模板方法进行查询（如下所示）：

```
int age = jdbcTemplate.queryForInt("select age from customer where customerId=?",new ➥
Object[]{new Integer(100)});
...
long interval = jdbcTemplate.queryForLong("select count(customerId) from customer");
...
String customerName = jdbcTemplate.queryForString("select username from customer where ➥
customerId=110");
...
Map singleCustomer = jdbcTemplate.queryForMap("select * from customer limit 1");
...
```

queryForMap方法与其他方法的不同之处在于，它的查询结果以java.util.Map的形式返回，Map的键对应所查询表的列名，Map的值是对应键所在列的值。当然，我们也看到了，这组模板方法主要用于单一结果的查询，使用的时候也请确保SQL查询所返回的结果是单一的，否则，JdbcTemplate将抛出org.springframework.dao.IncorrectResultSizeDataAccessException异常。

如果查询的结果将返回多行，而我们又不在乎它们是否拥有较强的类型约束，那么以下模板方法可以帮助我们：

- List queryForList(String sql)
- List queryForList(String sql, Object[] args)

queryForList方法根据传入的SQL以及相应的参数执行查询，将查询的结果以java.util.List的形式返回，返回的java.util.List中的每个元素都是java.util.Map类型，分别对应结果集中的一行，Map的键为每列的列名，而Map的值就是当前行列名对应的值。

如果这些还不足以满足我们的查询需要，那么就更进一步，使用相应的Callback接口对查询结果的返回进行定制。

用于查询的回调接口定义主要有以下三种。

org.springframework.jdbc.core.ResultSetExtractor。基本上属于JdbcTemplate内部使用的Callback接口。相对于下面两个Callback接口来说，ResultSetExtractor拥有更多的控制权，因为使用它，我们需要自行处理ResultSet。该接口定义如下：

```
public interface ResultSetExtractor {
    Object extractData(ResultSet rs) throws SQLException, DataAccessException;
}
```

在直接处理完ResultSet之后，我们可以将处理后的结果以任何想要的形式包装后返回。

org.springframework.jdbc.core.RowCallbackHandler 。RowCallbackHandler相对于ResultSetExtractor来说，只关注单行结果的处理，处理后的结果可以根据需要存放到当前RowCallbackHandler对象内，或者使用JdbcTemplate的程序上下文中。当然，这个完全是看个人喜好了。RowCallbackHandler的定义如下：

```
public interface RowCallbackHandler {
    void processRow(ResultSet rs) throws SQLException;
}
```

org.springframework.jdbc.core.RowMapper。ResultSetExtractor的精简版，功能类似于RowCallbackHandler，也只关注处理单行的结果。不过，处理后的结果会由ResultSetExtractor实现类进行组合。RowMapper的接口定义如下：

```
public interface RowMapper {
    Object mapRow(ResultSet rs, int rowNum) throws SQLException;
}
```

为了说明这三种Callback接口的使用和相互之间的区别，我们暂且设定如下场景：

数据库表customer中存在多行信息，对该表查询后，我们需要将每行的顾客信息都映射到域对象Customer中，并以java.util.List的形式返回所有的查询结果。

现在，我们分别使用这三种Callback接口对customer表进行查询，见代码清单14-16。

代码清单14-16　使用不同的回调接口处理查询结果的代码示例

```
List customerList = (List)jdbcTemplate.query("select * from customer", new ResultSetExtractor(){
    public Object extractData(ResultSet rs) throws SQLException,DataAccessException
    {
        List customers = new ArrayList();
        while(rs.next())
        {
            Customer customer = new Customer();
            customer.setFirstName(rs.getString(1));
            customer.setLastName(rs.getString(2));
            ...
            customers.add(customer);
        }
        return customers;
    }});

List customerList = jdbcTemplate.query("select * from customer", new RowMapper(){

    public Object mapRow(ResultSet rs, int rowNumber) throws SQLException {
        Customer customer = new Customer();
        customer.setFirstName(rs.getString(1));
        customer.setLastName(rs.getString(2));
        ...
        return customer;
    }});

final List customerList = new ArrayList();
jdbcTemplate.query("select * from customer", new RowCallbackHandler(){

    public void processRow(ResultSet rs) throws SQLException {
        Customer customer = new Customer();
        customer.setFirstName(rs.getString(1));
        customer.setLastName(rs.getString(2));
```

```
        ...
        customerList.add(customer);
    }});
```

如果你没有发现最大的差异在哪里，那么容我细表。

- 使用三种Callback接口作为参数的query方法的返回值不同：
  - 以ResultSetExtractor作为方法参数的query方法返回Object型结果，要使用查询结果，我们需要对其进行强制转型；
  - 以RowMapper接口作为方法参数的query方法直接返回List型的结果；
  - 以RowCallbackHandler作为方法参数的query方法，返回值为void。
- 使用ResultSetExtractor作为Callback接口处理查询结果，我们需要自己声明集合类，自己遍历ResultSet，自己根据每行数据组装Customer对象，自己将组装后的Customer对象添加到集合类中，方法最终只负责将组装完成的集合返回。
- 使用RowMapper比直接使用ResultSetExtractor要方便得多，只负责处理单行结果就行。现在，我们只需要将单行的结果组装后返回就行，剩下的工作，全部是JdbcTemplate内部的事情了。实际上，JdbcTemplae内部会使用一个ResultSetExtractor实现类来做其余的工作。

  JdbcTemplae内部使用的这个ResultSetExtractor实现类为org.springframework.jdbc.core.RowMapperResultSetExtractor，它内部持有一个RowMapper实例的引用。当处理结果集的时候，会将单行数据的处理委派给其所持有的RowMapper实例，而其余工作它负责，如以下代码所示：

  ```
  public Object extractData(ResultSet rs) throws SQLException {
      List results = (this.rowsExpected > 0 ? new ArrayList(this.rowsExpected) : new ArrayList());
      int rowNum = 0;
      while (rs.next()) {
          results.add(this.rowMapper.mapRow(rs, rowNum++));
      }
      return results;
  }
  ```

  这下应该清楚RowMapper为什么就处理单行结果就能完成ResultSetExtractor颇费周折的工作了吧？
- 虽然RowCallbackHandler与RowMapper同是处理单行数据，但是除了要处理单行结果，它还得负责最终结果的组装和获取工作，在这里我们使用当前上下文声明的List取得最终查询结果。不过，我们也可以单独声明一个RowCallbackHandler实现类，在其中声明相应的集合类，这样，可以通过该RowCallbackHandler实现类取得最终查询结果。代码清单14-17演示了这种用法。

代码清单14-17 单独声明的RowCallbackHandler使用示例

```
public class GenericRowCallbackHandler implements RowCallbackHandler {
    private List collections = new ArrayList();

    public void processRow(ResultSet rs) throws SQLException {
        Customer customer = new Customer();
        customer.setFirstName(rs.getString(1));
        customer.setLastName(rs.getString(2));
        ...
        collections.add(customer);
    }
```

```
        public List getResults()
        {
            return collections;
        }
    }

    GenericRowCallbackHandler handler = new GenericRowCallbackHandler();
    jdbcTemplate.query("select * from customer",handler);
    List customerList = handler.getResults();
```

该使用方式看起来是很明了了，不过GenericRowCallbackHandler的重用性却不佳。

RowCallbackHandler也是处理单行数据，那么谁来做遍历ResultSet的工作呢？RowCallbackHandlerResultSetExtractor，它其实也是一个ResultSetExtractor实现类，是JdbcTemplate一个内部静态类，一看它的定义你就知道奥秘之所在了（其定义见代码清单14-18）。

代码清单14-18 RowCallbackHandlerResultSetExtractor定义代码摘录

```
private static class RowCallbackHandlerResultSetExtractor implements ResultSetExtractor {

    private final RowCallbackHandler rch;

    public RowCallbackHandlerResultSetExtractor(RowCallbackHandler rch) {
        this.rch = rch;
    }

    public Object extractData(ResultSet rs) throws SQLException {
        while (rs.next()) {
                    this.rch.processRow(rs);
        }
        return null;
    }
}
```

总地来说，归根结底内部工作是由ResultSetExtractor做，RowCallbackHandler和RowMapper只是为了帮助我们简化使用上的操作而已。所以，在实际使用中，RowCallbackHandler和RowMapper才是最常用的选择。

使用JdbcTemplate进行查询，基本上就这些内容。而如果你非要使用基于StatementCallback之类更底层的execute方法的话，那就是你个人说了算啦。不过，要想知道JdbcTemplate中有关查询相关模板方法的更多信息，在实际使用中参考JdbcTemplate的Javadoc就可以。当然，有IDE就更便捷了。

基于JdbcTemplate的数据更新

相对于查询来说，使用JdbcTemplate进行数据更新就没有那么多说道了。不管是要对数据库进行数据插入，还是更新甚至删除，我们都可以通过JdbcTemplate所提供的一组重载的update()模板方法进行，这些update方法包括：

- int update(String sql)
- int update(String sql, Object[] args)
- int update(String sql, Object[] args, int[] argTypes)

下面是一些update()方法的使用样例：

```
// 插入数据
jdbcTemplate.update("insert into customer(customerName,age,...) values("darren","28",...)");
```

```
// 更新数据
int affectedRows = jdbcTemplate.update("update customer set customerName='daniel',age=36 ➥
where customerId=101");
// 或者
int affectedRows = jdbcTemplate.update("update customer set customerName=?,age=? where ➥
customerId=?", new Object[]{"Daniel",new Integer(36),new Integer(101)});
// 删除数据
int deletedRowCount = jdbcTemplate.update("delete from customer where customerId between 1 ➥
and 100");
```

通常情况下，接受简单的SQL以及相关参数的update方法就能够满足数据更新的需要。不过，如果我们觉得有必要对更新操作有更多的控制权，那么，可以使用与PreparedStatement相关的Callback接口。这包括使用PreparedStatementCreator创建PreparedStatement，使用PreparedStatementSetter对相关占位符进行设置等。同样的对一条记录进行更新，使用Callback接口作为参数的update方法的数据访问代码如代码清单14-19所示。

代码清单14-19　使用相应Callback接口进行数据更新的代码示例

```
// 使用int update(String sql, PreparedStatementSetter pss)方法
int affectedRows = jdbcTemplate.update("update customer set customerName=?,age=? where ➥
customerId=?",
new PreparedStatementSetter(){
    public void setValues(PreparedStatement ps) throws SQLException {
        ps.setString(1,"Daniel");
        ps.setInt(2,36);
        ps.setInt(3,101);
        }});

// 使用int update(PreparedStatementCreator psc)方法
int affectedRows = jdbcTemplate.update(new PreparedStatementCreator(){
    public PreparedStatement createPreparedStatement(Connection con) throws SQLException {
        PreparedStatement ps = con.prepareStatement("update customer set customerName=?,➥
        age=? where customerId=?");
        ps.setString(1,"Daniel");
        ps.setInt(2,36);
        ps.setInt(3,101);
        return ps;
        }});
```

使用update方法进行数据更新，可以获得最终更新操作所影响的记录数目，而且，如果不单单指定一个SQL作为参数的话，JdbcTemplate内部会构造相应的PreparedStatement进行实际的更新操作。

不过，除了使用update方法，我们还可以通过只接受SQL语句作为参数的execute()方法进行数据更新。该方法没有返回值，更加适合那种不需要返回值的操作，比如删除表、创建表等操作。如下代码给出了针对类似场景的代码示例：

```
jdbcTemplate.execute("create table customer (...)");
// 或者
jdbcTemplate.execute("drop table customer");
```

至于其他重载的execute()方法，相对来说过于贴近JDBC API了，通常情况下，我们没有必要使用。某些时候为了集成遗留系统中某些基于JDBC的数据访问代码，倒是有可能需要求助于这些execute方法。

批量更新数据

对于“更新同一数据表”的多笔更新操作来说，我们可以使用JDBC的批量更新（Batch Update）

功能对这些更新操作进行统一提交执行，以避免每一笔更新都单独执行，这样可以大大提高更新的执行效率。

JdbcTemplate提供了如下两个重载的batchUpdate()方法支持批量更新操作。

- int[] batchUpdate(String[] sql)
- int[] batchUpdate(String sql, BatchPreparedStatementSetter pss)

这两个方法在执行批量更新之前，会首先检查使用的JDBC驱动程序是否支持批量更新的功能，如果支持，则进行正常的批量更新；如果驱动程序不支持该功能，则会单独执行每一笔更新操作。

假设我们要将传入的新增加顾客列表的信息增加到数据库，那么就可以使用JdbcTemplate的批量更新支持来完成这一任务（详情见代码清单14-20）。

代码清单14-20 使用JdbcTemplate进行数据批量更新代码示例

```
public int[] insertNewCustomers(final List customers)
{
    jdbcTemplate.batchUpdate("insert into customer value(?,?,...)", new ➥
    BatchPreparedStatementSetter(){

    public int getBatchSize() {
        return customers.size();
    }

    public void setValues(PreparedStatement ps, int i) throws SQLException {
        Customer customer = (Customer)customers.get(i);
        ps.setString(1,customer.getFirstName());
        ps.setString(2,customer.getLastName());
        ...
        }});
}
```

因为更新语句中涉及参数，所以，使用BatchPreparedStatementSetter回调接口来对批量更新中每次更新所需要的参数进行设置。BatchPreparedStatementSetter有如下两个方法需要我们实现。

- **int getBatchSize()**。返回批量更新的数目，因为我们要对通过List传入的所有顾客信息进行更新，所以，当前批量更新的数目就是当前List中所有的顾客数目。
- **void setValues(PreparedStatement ps, int i)**。设置具体的更新数据，其中第二个int型的参数对应的是每笔更新的索引，我们就是根据这个索引从customers列表中取得相应的信息进行设置的。

Spring的批量更新相对于直接使用JDBC会有微小的性能损失。不过，当某些极端情况下（每个事务100万更新），使用Spring的批量更新可以取得很好的性能。

调用存储过程

存储过程是定义于数据库服务器端的计算单元。对于涉及多表数据而只使用SQL无法完成的计算，我们可以通过在数据库服务器端编写并部署存储过程的方式来实现。相对于将这些计算逻辑转移到客户端进行，使用存储过程的好处在于，可以避免像客户端计算那样在网络间来回传送数据导致的性能损失，因为存储过程的所有计算全部在服务器端完成。如果计算涉及多个数据表、大量的数据查询和更新，那么使用存储过程代替客户端计算是比较合适的做法。

存储过程（Stored Procedure）不是核心SQL标准的一部分，所以，并非所有关系数据库都提供对存储过程的支持。但存储过程在许多企业应用中具有重要地位，所以，JDBC标准也通过提供CallableStatement支持对现有存储过程的调用。

假设我们有以下的存储过程（MySQL的定义语法）：

```
CREATE PROCEDURE CountTable(IN tableName varchar(1000),OUT sqlStr varchar(1000) , INOUT v INT)
BEGIN
    set @flag = v;
    set @sql = CONCAT('select count(*) into @res from ' , tableName , ' where ACTIVE_FLAG=?');
    PREPARE stmt FROM @sql;
    EXECUTE stmt using @flag;
    DEALLOCATE PREPARE stmt;
    set v = @res;
    set sqlStr = @sql;
END
```

该存储过程定义了3个参数：

- tableName为IN参数，字符串类型；
- sqlStr为OUT参数，也是字符串类型；
- v为INOUT参数，INT类型；

对存储过程的调用基本上就是设置相应的IN参数，声明OUT参数，在CallableStatement执行后，再根据OUT参数取得执行结果这么一个流程。代码清单14-21演示的是没有使用JdbcTemplate，直接使用JDBC调用存储过程的代码示例。

代码清单14-21 直接使用JDBC API调用存储过程的代码示例

```
Connection conn = null;
CallableStatement stat = null;
try{
  conn = dataSource.getConnection();
  stat = conn.prepareCall("call CountTable(?,?,?)");

  stat.setString(1, "TableName");
  stat.setInt(3, 1);

  stat.registerOutParameter(2, Types.VARCHAR);
  stat.registerOutParameter(3, Types.INTEGER);

  stat.execute();

  String sql = stat.getString(2);
  int count = stat.getInt(3);

  System.out.println("SQL:"+sql);
  System.out.println("Record count:"+count);

 }
  catch(Exception dx){
  dx.printStackTrace();
 }
  finally{
    if(null!=stat) try{stat.close();}catch(Exception dx){}
    if(null!=conn) try{conn.close();}catch(Exception dx){}
}
```

JdbcTemplate同样对存储过程的调用进行了模板化处理，对于同一存储过程，我们来看使用JdbcTemplate后是怎么一个样子（见代码清单14-22）。

代码清单14-22 使用JdcTemplaet调用存储过程的代码示例

```
Object result = jdbcTemplate.execute("call CountTable(?,?,?)", new CallableStatementCallback(){
```

```
        public Object doInCallableStatement(CallableStatement cs)
                throws SQLException, DataAccessException {
            // 声明并设置IN/OUT参数
            cs.setString(1, "tableName");
            cs.setInt(3, 1);
            cs.registerOutParameter(2, Types.VARCHAR);
            cs.registerOutParameter(3, Types.INTEGER);

            // 执行Callablestatement
            cs.execute();

            // 抽取结果并返回
            Map result = new HashMap();
            result.put("SQL", cs.getString(2));
            result.put("COUNT", cs.getInt(3));
            return result;
        }});
```

我们直接使用CallableStatementCallback回调接口所公开的CallableStatement对象句柄进行调用操作，而无需关心CallableStatement以及Connection等资源的管理问题。

或者我们可以把CallableStatementCallback的部分职能划分出去，一部分由Callable-StatementCreator这个Callback接口分担，如代码清单14-23所示。

代码清单14-23　使用多个CallableStatement相关回调接口进行存储过程调用的代码示例

```
Object result = jdbcTemplate.execute(new CallableStatementCreator(){
    public CallableStatement createCallableStatement(Connection con) throws SQLException {
    CallableStatement cs = con.prepareCall("call CountTable(?,?,?)");
    cs.setString(1, "tableName");
    cs.setInt(3, 1);
    cs.registerOutParameter(2, Types.VARCHAR);
    cs.registerOutParameter(3, Types.INTEGER);
    return cs;
    }}, new CallableStatementCallback(){
    public Object doInCallableStatement(CallableStatement cs) ➥
    throws SQLException, DataAccessException {
    cs.execute();
    // 抽取结果并返回
    Map result = new HashMap();
    result.put("SQL", cs.getString(2));
    result.put("COUNT", cs.getInt(3));
    return result;
    }});
```

除了以上两种调用存储过程的方法，我们还可以使用JdbcTemplate提供的另一个调用存储过程的模板方法，如下：

```
Map call(CallableStatementCreator csc, List declaredParameters)
```

该模板方法的主要好处是可以通过List指定存储过程的参数列表，之后，JdbcTemplate会根据指定的参数列表所提供的参数信息为我们组装调用结果，并以Map形式返回。declaredParameters参数列表中的元素，需要是org.springframework.jdbc.core.SqlParameter类型或者相关子类，声明顺序和参数类型要与实际存储过程定义的参数顺序和类型相同。

代码清单14-24给出的是使用call方法调用我们的存储过程的实例代码。

代码清单14-24　使用JdbcTemplate的call模板方法调用存储过程的代码示例

```
List<SqlParameter> parameters = new ArrayList<SqlParameter>();
```

```
parameters.add(new SqlParameter(Types.VARCHAR));
parameters.add(new SqlOutParameter("SQL",Types.VARCHAR));
parameters.add(new SqlInOutParameter("COUNT",Types.INTEGER));

Map result = jdbcTemplate.call(new CallableStatementCreator(){

    public CallableStatement createCallableStatement(Connection con) ➥
    throws SQLException {
        CallableStatement cs = con.prepareCall("call CountTable(?,?,?)");
        cs.setString(1, "tableName");
        cs.setInt(3, 1);
        cs.registerOutParameter(2, Types.VARCHAR);
        cs.registerOutParameter(3, Types.INTEGER);
        return cs;
    }}, parameters);

System.out.println(result.get("SQL"));
System.out.println(result.get("COUNT"));
```

使用JdbcTemplate的存储过程调用方法，我们不用关注资源释放之类的问题，仅关注相关参数和结果的处理即可。虽然可以省却资源管理的烦恼，但使用相关回调接口使得使用JdbcTemplate进行存储过程调用并不是那么令人赏心悦目。如果你不满意此种烦琐，那么没关系，稍后将介绍Spring中另一种存储过程调用方式。

● 递增主键生成策略的抽象

在为关系数据库增加数据时，新增数据的主键生成一直是一个需要关注的问题。对于数据主键的生成位置，通常有两种选择：第一种选择是在数据库服务器端，使用不同的数据库厂商提供的主键生成策略支持，第二种选择是直接在应用程序的客户端，根据某些算法生成需要的数据主键。虽然第一种选择可以充分利用数据库的特性以及优化措施，但可移植性比较差，而且某些情况下可能造成数据库的过多负担；采用客户端的主键生成策略（即第二种选择），可以分担服务器的负担，而且主键的生成策略可以根据情况进行调整，灵活性很好，性能也可能随着系统架构的不同而有所提升。

在设计某些应用程序的时候，出于主键生成策略的可移植性或者性能方面的考虑，会在应用程序客户端采用某种主键生成抽象策略，以统一的方式进行主键生成的管理。大部分情况下，递增的主键生成策略是我们在这种情况下使用最多的主键生成策略。我想，在此之前，你我或多或少都做过同样的事情。起码，我经历的FX项目就采用这种类似的主键生成策略。现在，Spring对递增的主键生成策略进行了适当的抽象，针对不同的关系数据库给出了相应的主键生成实现类，帮助我们统一基于递增策略的主键生成。

Spring的org.springframework.jdbc.support.incrementer包下是针对递增主键生成策略的相关接口定义和实现类，org.springframework.jdbc.support.incrementer.DataFieldMaxValueIncrementer是这个体系的顶层接口定义，其定义如下：

```
public interface DataFieldMaxValueIncrementer {
    int nextIntValue() throws DataAccessException;
    long nextLongValue() throws DataAccessException;
    String nextStringValue() throws DataAccessException;
}
```

根据不同数据库对递增主键生成的支持，DataFieldMaxValueIncrementer的相关实现类可以分为如下两类。

- 基于独立主键表的DataFieldMaxValueIncrementer。基于独立主键表的DataFieldMaxValueIncrementer依赖于为每一个数据表单独定义的主键表，主键表中定义的主键可以根据

需要获取并递增，并且可以设置每次获取的CacheSize以减少访问数据库资源的频度。Spring为HSQLDB和MySQL数据库提供的就是基于这种策略的DataFieldMaxValueIncrementer实现。

- 基于数据库Sequence的DataFieldMaxValueIncrementer。像DB2或者Oracle等数据库，数据库本身支持基于Sequence的主键生成，所以，Spring在数据库本身的Sequence的基础上，为DB2、Oracle和PostgreSQL等支持Sequence的数据库，提供了基于Sequence的DataFieldMaxValue-Incrementer实现类。

整个DataFieldMaxValueIncrementer的继承层次图如图14-3所示。

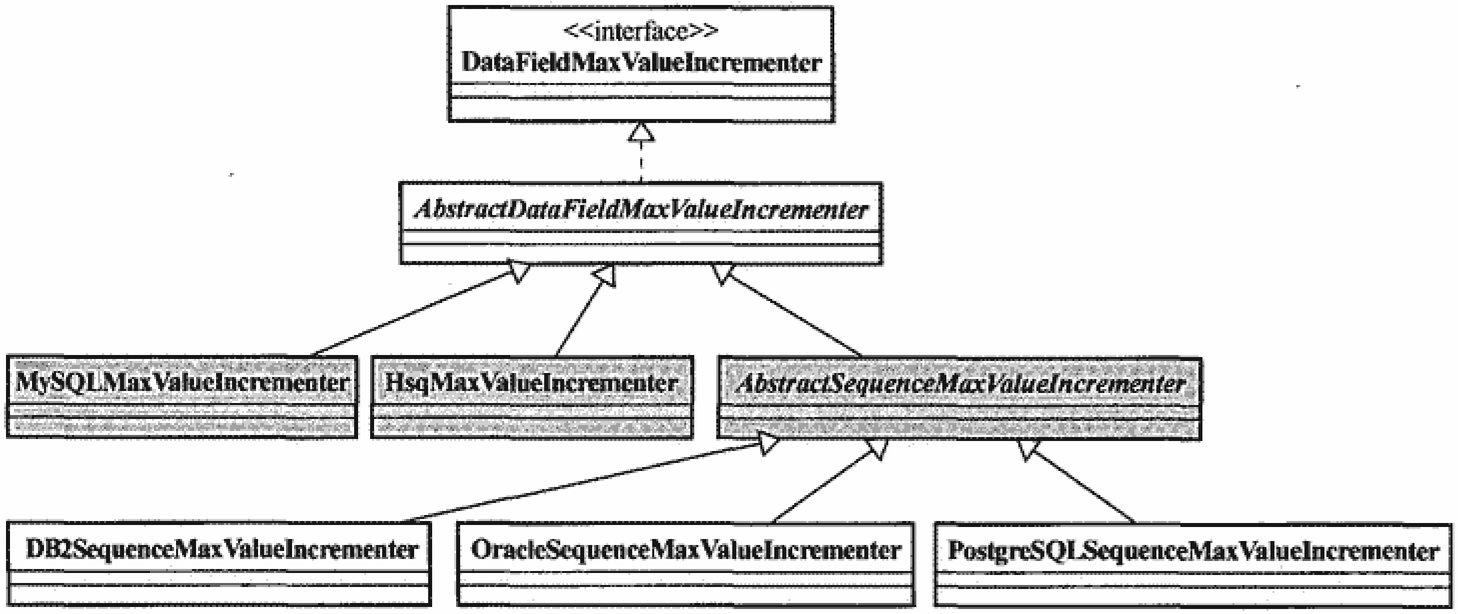

图14-3 DataFieldMaxValueIncrementer继承层次图

基于独立主键表的DataFieldMaxValueIncrementer

HsqlMaxValueIncrementer和MySQLMaxValueIncrementer继承自AbstractDataFieldMax-ValueIncrementer，是依赖主键定义表的DataFieldMaxValueIncrementer实现，要使用这两种DataFieldMaxValueIncrementer实现，我们需要为相应的表定义对应的主键表，以保存相应的主键值。

还记得在第2章中提到的FX News吗？假设我们采用如下代码中的表定义来保存各种新闻内容：

```
CREATE TABLE fx_news (
    news_id      bigint(20) NOT NULL,
    new_title    varchar(25) NOT NULL,
    new_body     text NOT NULL,
    PRIMARY KEY(news_id)
)
```


news_id为该表的主键，要使用MySQLMaxValueIncrementer（或者HsqlMaxValueIncrementer）在每次插入数据的时候为该字段生成主键值，还需要定义对应fx_news表的主键表，用来计算并保持当前主键值，对应的fx_news主键表定义如下：

```
CREATE TABLE fx_news_key (
    value    bigint(20)  NOT NULL default 0,
    PRIMARY KEY(value)
) engine=MYISAM;
```

```
insert into fx_news_key values(0);
```

注意，为了减少事务开销，我们将fx_news_key主键表的引擎设置为MYISAM，而不是InnoDB。

有了这些，我们就可以在应用程序中使用MySQLMaxValueIncrementer生成递增主键，并向关系数据库插入新增数据了，如以下代码所示：

```
DataSource dataSource = ...;
JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
DataFieldMaxValueIncrementer incrementer = new MySQLMaxValueIncrementer ➥
(dataSource,"fx_news_key","value");
((MySQLMaxValueIncrementer)incrementer).setCacheSize(5);
jdbcTemplate.update("insert into fx_news(news_id,news_title,news_body) values(?,?,?)",➥
new Object[]{incrementer.nextLongValue(),"title","body"});
```

我们需要为MySQLMaxValueIncrementer提供一个DataSource，以及对应主键表的表名和相应的列。如果需要，我们还可以指定CacheSize，以便每次取得多个值在本地缓存，从而减少数据库访问次数。可以通过设置cacheSize进行本地的主键值缓存是MySQLMaxValueIncrementer和HsqlMaxValueIncrementer比较实用的一个特色。

通常情况下，使用Spring的IoC容器来配置相应的DataFieldMaxValueIncrementer要方便得多。这样，在整个应用程序中，那些需要递增主键生成的类，都可以很容易地获取DataFieldMaxValueIncrementer相应实现类的注入，代码清单14-25给出了一个常见的配置示例。

代码清单14-25　容器中DataFieldMaxValueIncrementer的配置使用示例

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
    <property name="url">
        <value>${jdbc.url}</value>
    </property>
    <property name="driverClassName">
        <value>${jdbc.driver}</value>
    </property>
    <property name="username">
        <value>${jdbc.username}</value>
    </property>
    <property name="password">
        <value>${jdbc.password}</value>
    </property>
</bean>

<bean id="incrementer" class="org.springframework.jdbc.support.incrementer. ➥
MySQLMaxValueIncrementer">
    <property name="dataSource" ref="dataSource"/>
    <property name="incrementerName" value="fx_news_key"/>
    <property name="columnName" value="value"/>
</bean>

<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
    <property name="dataSource" ref="dataSource"/>
</bean>

<bean id="djNewsPersister" class="...DowJonesNewsPersister">
    <property name="incrementer" ref="incrementer"/>
    <property name="jdbcTemplate" ref="jdbcTemplate"/>
</bean>
```

每次DowJonesNewsPersister向数据库增加新闻数据的时候，就可以使用为其注入的incre-

menter递增主键了。

使用IoC容器来管理相应的DataFieldMaxValueIncrementer，可以将系统中所有的DataFieldMaxValueIncrementer实例集中到一个配置模块中，以便于管理和使用。不过，使用容器管理的DataFieldMaxValueIncrementer可能需要注意一个问题，那就是，虽然容器中的各个DataFieldMaxValueIncrementer在系统中可以共享，但即使在系统不使用的情况下，相应的实例也不会释放，除非系统推出。所以，对于系统资源紧张的应用来说，在合适的时机，根据需要实例化相应的DataFieldMaxValueIncrementer来使用，也不失为合适的方式。

基于数据库Sequence的DataFieldMaxValueIncrementer

对于提供了Sequence支持的数据库来说，DataFieldMaxValueIncrementer实现体系专门在AbstractDataFieldMaxValueIncrementer的基础上，开辟了一个面向基于Sequence的DataFieldMaxValueIncrementer实现分支，以AbstractSequenceMaxValueIncrementer作为这个分支的统一超类。Spring为支持Sequence的DB2、Oracle和PostgreSQL数据库提供了相应的DataFieldMaxValueIncrementer实现类：

- DB2SequenceMaxValueIncrementer
- OracleSequenceMaxValueIncrementer
- PostgreSQLSequenceMaxValueIncrementer

要使用基于Sequence的DataFieldMaxValueIncrementer，只要数据库中定义了相应的Sequence就可以。以Oracle数据库为例，同样针对fx_news表，我们现在不使用主键表，而是为其定义一个Sequence如下：

```
CREATE  SEQUENCE fx_news_seq
NCREMENT  BY   1
START  WITH   1
NOMAXVALUE
NOCYCLE
NOCACHE;
```

之后，我们只要在构造OracleSequenceMaxValueIncrementer的时候，告知对应的Sequence名称即可，如下所示：

```
DataSource dataSource = ...;
JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
DataFieldMaxValueIncrementer incrementer = new ➥
OracleSequenceMaxValueIncrementer(dataSource,"fx_news_seq");
jdbcTemplate.update("insert into fx_news(news_id,news_title,news_body) values(?,?,?)",➥
new Object[]{incrementer.nextLongValue(),"title","body"});
```

如果想每次取得一批数据，那么需要在Sequence的定义中指定，而无法在客户端调用的时候决定是每次取一个数据还是一批数据，例如：

```
CREATE  SEQUENCE fx_news_seq
NCREMENT  BY   1
START  WITH   1
NOMAXVALUE
NOCYCLE
CACHE 5;
```

当然，如果应用程序构建在Spring的IoC容器上，从基于主键表的DataFieldMaxValueIncrementer实现类变为基于Sequence的DataFieldMaxValueIncrementer实现类，只是简单的配置

更改而已，见代码清单14-26。

代码清单14-26　基于Sequence的DataFieldMaxValueIncrementer容器内配置使用示例

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method= ➥
  "close">
    ...
</bean>

<bean id="incrementer" class="org.springframework.jdbc.support.incrementer. ➥
OracleSequenceMaxValueIncrementer">
    <property name="dataSource" ref="dataSource"/>
    <property name="incrementerName" value="fx_news_seq"/>
</bean>

...

<bean id="djNewsPersister" class="...DowJonesNewsPersister">
    <property name="incrementer" ref="incrementer"/>
    <property name="jdbcTemplate" ref="jdbcTemplate"/>
</bean>
```

其他的bean定义和使用DataFieldMaxValueIncrementer的类逻辑基本不用动。

总之，如果需要递增的主键生成策略，根据应用程序使用的数据库选用相应的DataFieldMaxValueIncrementer实现类即可。如果Spring框架没有提供针对当前应用程序所用数据库的DataFieldMaxValueIncrementer实现类，那么不妨在AbstractDataFieldMaxValueIncrementer或者AbstractSequenceMaxValueIncrementer的基础上扩展一个啦！

- **Spring中的LOB类型处理**

LOB（Large OBject），指的是数据库中能够存取大量数据的数据类型[①]。按照存放的具体数据形式，LOB类型通常分为BLOB和CLOB两种类型[②]。

- BLOB（Binary Large OBject），主要用于存放数据量比较大的二进制类型数据，如比较大的图像、Word文档之类的二进制文件。
- CLOB（Character Large Object），主要用于存放数据量比较大的文本类型数据。

因为LOB类型针对的是大数据量数据，所以在处理数据的方式上与处理其他类型数据不同。虽然JDBC标准对LOB类型数据操作已经进行了标准化的规定，但是，或许是历史性原因，某些数据库在处理LOB数据的时候，需要使用数据库特定的API才能处理，比如Oracle。通常情况下，我们的应用程序数据访问逻辑需要根据后台的数据库类型进行相应调整。

除了Oracle，其他数据库在进行LOB字段的更新和读取的时候，与通常的字段类型没有太多差别，大都是通过PreparedStatement的相应方法设置LOB的值，然后通过ResultSet的相应方法获取结果集中的LOB数据。假设，我们有如下的表定义用于存放图像文件的相关数据（以MySQL方式定义）：

```
CREATE TABLE images (
    id          int(11)  NOT NULL,
    filename    varchar(200) NOT NULL,
```

① Oracle 9i数据库中Lob类型可以存储的数据量可以高达4 GB。Oracle 10g中LOB类型的容量甚至可以达到8 TB到128 TB之间。当然，这要取决于数据库是如何进行配置。

② 实际上，如果对LOB类型细分的话，具体类型要多得多。比如，Oracle中就有BLOB、CLOB、NCLOB、BFILE等，其他数据库也有各种对应LOB的类型定义。只能说，BLOB和CLOB是比较有代表性的LOB类型。

```
    entity          blob NOT NULL,
    description     text NULL,
    PRIMARY KEY(id)
)
```

只要不是Oracle数据库，我们就可以如代码清单14-27所示，按照通常的JDBC操作方式对BLOB类型进行更新和读取（忽略异常处理）。

代码清单14-27 LOB数据类型操作代码示例

```
// --- 将文件数据以二进制流的形式存入BLOB ---
File imageFile = new File("snow_image.jpg");
InputStream ins = new FileInputStream(imageFile);

Connection con = dataSource.getConnection();
PreparedStatement ps = con.prepareStatement("insert into ➥
images(id,filename,entity,description) values(?,?,?,?)");
ps.setInt(1, 1);
ps.setString(2, "snow_image.jpg");
ps.setBinaryStream(3, ins,(int)imageFile.length());
ps.setString(4, "nothing to say");
ps.executeUpdate();
ps.close();
con.close();

IOUtils.closeQuietly(ins);
...

//  --- 以二进制流的形式读取数据 ---
File imageFile = new File("snow_image_copy.jpg");
InputStream ins = null;

Connection con = dataSource.getConnection();
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("select entity from images where id=1");
while(rs.next())
{
    ins = rs.getBinaryStream(1);
}
rs.close();
stmt.close();
con.close();

OutputStream ous = new FileOutputStream(imageFile);
IOUtils.write(IOUtils.toByteArray(ins), ous);
IOUtils.closeQuietly(ins);
IOUtils.closeQuietly(ous);
```

14

在这里，我们直接使用PreparedStatement的setBinaryStream()方法，对BLOB类型数据进行存储，使用ResultSet的getBinaryStream()方法对BLOB数据进行读取（也可以使用针对Object和byte[]类型的PreparedStatement的setXXX()方法或者ResultSet的getXXX()方法对BLOB数据进行存储，更多信息请参考JDBC文档）。

可是一旦使用Oracle数据库，麻烦就来了，我们只能通过Oracle驱动程序提供的oracle.sql.BLOB或者oracle.sql.CLOB实现类对LOB数据进行操作，而无法通过标准的JDBC API进行。在Oracle 9*i*

中，如果我们要同样地插入一笔数据，代码如代码清单14-28所示。

代码清单14-28　Oracle中使用JDBC进行LOB数据的操作代码示例

```
Connection con = ...;
Statement stmt = con.createStatement();
// 1. 要插入一笔BLOB数据，需要先插入empty blob以占位
stmt.executeUpdate("insert into images(id,filename,entity,description) ➥
values(1,'snow_image.jpg',empty_blob(),'no desc')");
// 2. 取回对应记录的BLOB的locator，然后通过locator写入数据
ResultSet rs = stmt.executeQuery("select entity from images where id=1");
rs.next();
BLOB blob = ((OracleResultSet)rs).getBLOB(1);

File imageFile = new File("snow_image.jpg");
InputStream ins = new FileInputStream(imageFile);
OutputStream ous = blob.getBinaryOutputStream();
IOUtils.write(IOUtils.toByteArray(ins), ous);
IOUtils.closeQuietly(ins);
IOUtils.closeQuietly(ous);

rs.close();
stmt.close();
con.close();
```

对于查询来说，也要通过oracle.sql.BLOB或者oracle.sql.CLOB实现类进行，如代码清单14-29所示。

代码清单14-29　Oracle中对LOB数据进行查询的操作代码示例

```
Connection con = ...;
Statement stmt = con.createStatement();

ResultSet rs = stmt.executeQuery ("select entity from images where id=1");
rs.next();
BLOB blob = ((OracleResultSet)rs).getBLOB(1);
//使用blob.getBinaryStream()或者getBytes()方法处理结果即可

rs.close();
stmt.close();
con.close();
```

鉴于对LOB数据处理方式的不一致性，Spring在org.springframework.jdbc.support.lob包下面，提出了一套LOB数据处理类，用于屏蔽各数据库驱动在处理LOB数据方式上的差异性。org.springframework.jdbc.support.lob.LobHandler接口是Spring框架得以屏蔽LOB数据处理差异性的核心，它只定义了对BLOB和CLOB数据的操作接口，而具体的实现则留给具体的实现类来做。我们可以通过LobHandler提供的各种BLOB和CLOB数据访问方法，以需要的方式对LOB进行读取。LobHandler定义见代码清单14-30。

代码清单14-30　LobHandler定义

```
public interface LobHandler {
    byte[] getBlobAsBytes(ResultSet rs, String columnName) throws SQLException;
    byte[] getBlobAsBytes(ResultSet rs, int columnIndex) throws SQLException;
    InputStream getBlobAsBinaryStream(ResultSet rs, String columnName) throws SQLException;
    InputStream getBlobAsBinaryStream(ResultSet rs, int columnIndex) throws SQLException;
    String getClobAsString(ResultSet rs, String columnName) throws SQLException;
```

```
    String getClobAsString(ResultSet rs, int columnIndex) throws SQLException;
    InputStream getClobAsAsciiStream(ResultSet rs, String columnName) throws SQLException;
    InputStream getClobAsAsciiStream(ResultSet rs, int columnIndex) throws SQLException;
    Reader getClobAsCharacterStream(ResultSet rs, String columnName) throws SQLException;
    Reader getClobAsCharacterStream(ResultSet rs, int columnIndex) throws SQLException;
    LobCreator getLobCreator();
}
```

LobHandler除了作为LOB数据的访问接口，它还有一个角色，那就是它还是org.springframework.jdbc.support.lob.LobCreator的生产工厂，从LobHandler定义的最后一行应该看得出来。LobCreator的职责主要在于LOB数据的创建，它让我们能够以统一的方式创建LOB数据。我们将在插入或者更新LOB数据的时候使用它。不过在此之前，我们先把LobCreator放一边，继续关注LobHandler。

LobHandler的继承关系如图14-4所示。

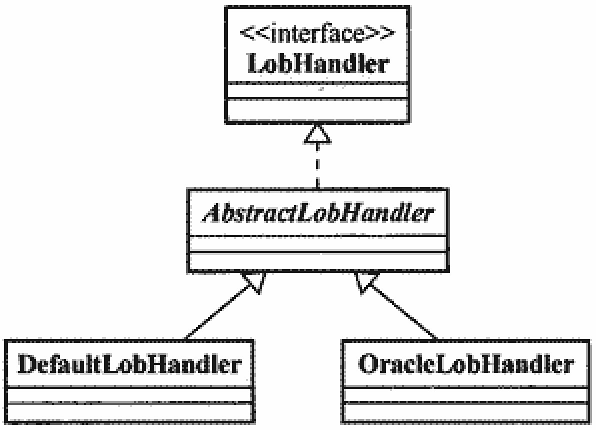

图14-4 LobHandler的继承关系

整个层次很简单，在LobHandler下实现了一个AbstractLobHandler抽象类以简化子类的实现。这个抽象类的逻辑很简单，就是将LobHandler中位于同一组的重载方法其中一个的逻辑委托给另一个，比如：

```
public abstract class AbstractLobHandler implements LobHandler {

    public byte[] getBlobAsBytes(ResultSet rs, String columnName) throws SQLException {
        return getBlobAsBytes(rs, rs.findColumn(columnName));
    }
    ...
}
```

所以，我们应该更多的关注OracleLobHandler和DefaultLobHandler这两个具体实现类。

- org.springframework.jdbc.support.lob.OracleLobHandler是专门针对Oracle数据库的LobHandler实现，通过Oracle数据库驱动的原始API进行LOB数据的操作。因为要用到驱动程序的原始API，所以，在使用OracleLobHandler的时候，我们需要根据情况为它提供相应的NativeJdbcExtractor实现。比如，如果我们通过Commons DBCP数据库缓冲池管理Oracle数据库连接的话，那么，在使用OracleLobHandler进行LOB数据处理之前，需要如下所示，为其设置CommonsDbcpNativeJdbcExtractor：

```
OracleLobHandler lobHandler = new OracleLobHandler();
lobHandler.setNativeJdbcExtractor(new CommonsDbcpNativeJdbcExtractor());
// 处于可以使用状态的lobHandler
```

- 除了Oracle之外的大多数数据库，可以使用DefaultLobHandler作为它们的LobHandler实现来操作LOB数据。DefaultLobHandler主要通过标准的JDBC API来创建和访问LOB数据。

现在，对images表的数据进行创建和访问就不会因为数据库的变动而做相应的调整了。要调整的只是选择使用哪个LobHandler实现类而已，而通常这只需要调整一行配置即可。

使用JdbcTemplate对LOB数据进行操作，我们通常使用org.springframework.jdbc.core.support.AbstractLobCreatingPreparedStatementCallback作为Callback接口，该类构造的时候接受一个LobHandler作为构造方法参数。

14

同样向images表插入一条数据，使用JdbcTemplate和AbstractLobCreatingPreparedStatementCallback之后的数据访问代码如代码清单14-31所示。

代码清单14-31 使用LobHandler和JdbcTemplate进行数据更新的代码示例

```
final File imageFile = new File("snow_image.jpg");
final InputStream ins = new FileInputStream(imageFile);
LobHandler lobHandler = new DefaultLobHandler();
jdbcTemplate.execute("insert into images(id,filename,entity,description) ➥
values(?,?,?,?)", new AbstractLobCreatingPreparedStatementCallback(lobHandler){

    @Override
    protected void setValues(PreparedStatement ps, LobCreator lobCreator) ➥
    throws SQLException, DataAccessException {
        ps.setInt(1, 2);
        ps.setString(2, "snow_image.jpg");
        lobCreator.setBlobAsBinaryStream(ps, 3, ins, (int)imageFile.length());
        ps.setString(4, "nothing to say");
    }});
IOUtils.closeQuietly(ins);
```

查询数据的访问代码则看起来如代码清单14-32所示。

代码清单14-32 使用LobHandler和JdbcTemplate进行数据查询的代码示例

```
final LobHandler lobHandler = new DefaultLobHandler();
InputStream ins = (InputStream)jdbcTemplate.queryForObject("select entity from images where ➥
id=1", new RowMapper(){
    public Object mapRow(ResultSet rs, int row) throws SQLException {
        return lobHandler.getBlobAsBinaryStream(rs, 1);
    }});

// 将LOB数据写入文件
File imageFile = new File("snow_image_copy.jpg");
OutputStream ous = new FileOutputStream(imageFile);
IOUtils.write(IOUtils.toByteArray(ins), ous);
IOUtils.closeQuietly(ins);
IOUtils.closeQuietly(ous);
```

对于查询来说，如果是特定于LOB结果的处理，通常使用org.springframework.jdbc.core.support.AbstractLobStreamingResultSetExtractor作为结果集的回调处理接口，如代码清单14-33所示。

代码清单14-33 使用AbstractLobStreamingResultSetExtractor处理LOB对象的代码示例

```
final OutputStream ous = new FileOutputStream(imageFile);
jdbcTemplate.query("select entity from images where id=1",➥
new AbstractLobStreamingResultSetExtractor(){
    @Override
    protected void streamData(ResultSet rs) throws SQLException, ➥
    IOException, DataAccessException {
        InputStream ins = lobHandler.getBlobAsBinaryStream(rs, 1);
        IOUtils.write(IOUtils.toByteArray(ins), ous);
        IOUtils.closeQuietly(ins);
    }});
IOUtils.closeQuietly(ous);
```

现在要对读取后的数据作何处理，直接在streamData方法中指定就可以了。

如果应用程序使用Spring的IoC容器，我们可以将LobHandler的定义追加的容器的配置文件中。如果因为数据库的变动需要变换LobHandler具体实现类，那也只需做简单的配置变更。所以，我强烈建议使用Spring的IoC容器管理整个应用的配置和运行。

2. **NamedParameterJdbcTemplate**

对于每次调用都需要动态指定查询或者更新参数的SQL来说，通常或者说自始至终，我们都是通过?作为SQL参数的占位符的。有了NamedParameterJdbcTemplate的支持之后，我们就可以通过容易记忆或者更加有语义的符号来作为SQL中的参数占位符。如果觉得使用语义符号作为占位符更好，那么我们就把?形式的占位符抛入到历史的尘埃中去。如果觉得两种占位符方式都还可以，那么我们也算有了更多的选择。

- NamedParameterJdbcTemplate的功能

让我们先看一下同一条SQL语句，使用?作为占位符和使用命名的参数符号作为占位符前后有何差别，如下所示：

```
// 前
select count(*) from images where filename=?
// 后
select count(*) from images where filename=:filename
```

:filename就是命名的参数符号（如果你用过Ruby，是不是会觉得很熟悉呢）。通过NamedParameterJdbcTemplate，我们就可以执行这种使用命名参数符号的SQL语句，如下所示：

```
DataSource dataSource = ...;
NamedParameterJdbcTemplate npJdbcTemplate = new NamedParameterJdbcTemplate(dataSource);

SqlParameterSource parameterSource = new MapSqlParameterSource("filename","snow_image.jpg");
int count = npJdbcTemplate.queryForInt("select count(*) from images where filename=:filename",➥
parameterSource);
// 根据需要处理或使用count结果
```

现在使用NamedParameterJdbcTemplate，不是通过Object[]数组的形式为相应占位符提供参数值，而是通过org.springframework.jdbc.core.namedparam.SqlParameterSource接口，该接口定义有两个实现类，分别是org.springframework.jdbc.core.namedparam.MapSql ParameterSource和org.springframework.jdbc.core.namedparam.BeanPropertySql ParameterSource。在上例中，我们已经使用了MapSqlParameterSource，它持有一个Map实例，所有的命名参数符号以及它们的值都存入这个持有的Map实例中。所以，如果SQL中存在不止一个命名参数符号的话，我们也可以通过MapSqlParameterSource的addValue(String, Object)方法添加多个命名参数符号及相关值，如下所示。

```
如果SQL是:
select count(*) from images where filename=:filename and description=:description

SqlParameterSource parameterSource = ➥
new MapSqlParameterSource("filename","snow_image.jpg");
parameterSource = parameterSource.addValue("description","something");
npJdbcTemplate.queryForInt(SQL, parameterSource);
```

因为MapSqlParameterSource实际上就是对Map的一个封装，所以，NamedParameterJdbcTemplate也提供了使用Map作为方法参数的重载的模板方法，我们也可以直接使用Map来替代相应的SqlParameterSource实例（不只MapSqlParameterSource），例如：

```
DataSource dataSource = ...;
NamedParameterJdbcTemplate npJdbcTemplate = new NamedParameterJdbcTemplate(dataSource);

Map parameters = new HashMap();
parameters.put("filename","snow_image.jpg");
parameters.put("description","something");

int count = npJdbcTemplate.queryForInt(SQL, parameters);
// 根据需要处理或使用count结果
```

SqlParameterSource的另一个实现类BeanPropertySqlParameterSource允许我们对bean对象进行封装，并使用相应的bean对象的属性值作为命名参数符号的值。如果images表对应的域对象（domain object）定义如代码清单14-34所示。

代码清单14-34 Image域对象定义

```
public class Image
{
    private int id;
    private String filename;
    private byte[] entity;
    private String decription;

    public String getFileName()
    {
        return this.filename;
    }

    public String getDescription()
    {
        return this.description;
    }
    // 其他setter和getter方法定义
    ...
}
```

那么，我们可以使用BeanPropertySqlParameterSource对其进行封装，然后作为参数传给NamedParameterJdbcTemplate进行数据访问，如下所示：

```
DataSource dataSource = ...;
NamedParameterJdbcTemplate npJdbcTemplate = new NamedParameterJdbcTemplate(dataSource);

Image image = new Image();
image.setFilename("snow_image.jpg");
image.setDescription("nothing to say");
SqlParameterSource parameterSource = new BeanPropertySqlParameterSource(image);

int count = npJdbcTemplate.queryForInt(SQL, parameterSource);
```

当然，对于使用BeanPropertySqlParameterSource的情况，最好是通过方法参数直接传入bean对象。要是在方法内部自己构造bean对象，然后又通过BeanPropertySqlParameterSource进行封装，显然是有些烦琐了。

使用BeanPropertySqlParameterSource唯一需要注意的就是，SQL中的命名参数符号的名称应

该与bean对象定义的属性名称一致。

● NamedParameterJdbcTemplate的实现原理

其实，在有了JdbcTemplate的基础之后，要实现NamedParameterJdbcTemplate就容易得多了，NamedParameterJdbcTemplate只需要提供JdbcTemplate之外的特性就可以。.

NamedParameterJdbcTemplate内部持有一个org.springframework.jdbc.core.JdbcOperations实例，JdbcTemplate是JdbcOperations的唯一实现类，所以，说NamedParameterJdbcTemplate内部有一个JdbcTemplate更直接一些。顺理成章的，如果我们已经有了一个JdbcTemplate的实例，而又想构造一个NamedParameterJdbcTemplate，那么除了可以传入相应的DataSource作为参数，也可以传入已有的JdbcTemplate作为参数，如下所示：

```
DataSource dataSource = ...;
NamedParameterJdbcTemplate npJdbcTemplate = new NamedParameterJdbcTemplate(dataSource);

// 或者已经存在一个JdbcTemplate实例
JdbcTemplate jdbcTemplate = ...;
NamedParameterJdbcTemplate npJdbcTemplate = new NamedParameterJdbcTemplate(jdbcTemplate);
```

要通过NamedParameterJdbcTemplate获取内部持有的JdbcTemplate实例，可以通过其getJdbcOperations()方法，如下所示：

```
NamedParameterJdbcTemplate npJdbcTemplate = new NamedParameterJdbcTemplate(jdbcTemplate);
JdbcOperations jdbcOper = npJdbcTemplate.getJdbcOperations();

assertSame(jdbcTemplate,jdbcOper);
```

NamedParameterJdbcTemplate的getJdbcOperations()方法实际上是其父接口org.springframework.jdbc.core.namedparam.NamedParameterJdbcOperations定义的。该接口定义的一组方法接受使用命名参数符号的SQL和SqlParameterSource或者Map作为方法参数，而具体的实现是由NamedParameterJdbcTemplate来做的。

NamedParameterJdbcTemplate的模板方法开始执行，首先借助于NamedParameterUtils工具类对传入的使用了命名参数符号的SQL进行解析，然后使用旧有的?占位符替换掉SQL中相应的命名参数符号，之后，根据替换后的SQL和从SqlParameterSource中解析出来的参数信息，直接调用内部持有的JdbcTemplate实例来执行更新或者查询操作即可。

基本上，NamedParameterJdbcTemplate的实现就是在JdbcTemplate基础上添加一层解析网。

3. SimpleJdbcTemplate

SimpleJdbcTemplate是Spring为构建在Java 5或者更高版本Java上的应用程序提供的更加便利的JdbcTemplate实现。要使用SimpleJdbcTemplate，需先确认你的Java版本是否符合要求。

SimpleJdbcTemplate集JdbcTemplate和NamedParameterJdbcTemplate的功能于一身，并且在这二者上添加了Java 5之后引入的动态参数（varargs）、自动拆箱解箱（autoboxing）和范型（generic）的支持。现在，我们可以使用动态参数的形式取代Object[]参数的形式，也可以利用自动拆箱解箱功能避免原始类型到相应封装类型的转换，又可以声明强类型的返回值类型，而不是只有Object。

代码清单14-35给出的是使用SimpleJdbcTemplate后的部分代码演示，从中我们能够看到Java 5之后的一些特色功能。

代码清单14-35 SimpleJdbcTemplate使用代码示例

```
DataSource dataSource = ...;
```

```
SimpleJdbcTemplate sjt = new SimpleJdbcTemplate(dataSource);
final LobHandler lobHandler = new DefaultLobHandler();
String SQL = "select * from images where filename=? and description=?";
ParameterizedRowMapper<Image> rowMapper = new ParameterizedRowMapper<Image>(){

    public Image mapRow(ResultSet rs, int row) throws SQLException {
        Image image = new Image();
        image.setId(rs.getInt(1));
        image.setFilename(rs.getString(2));
        image.setEntity(lobHandler.getBlobAsBytes(rs, 3));
        image.setDescription(rs.getString(4));
        return image;
    }
};

Image image = sjt.queryForObject(SQL, rowMapper, "snow_image.jpg","nothing to say");
System.out.println(image.getDescription());
```

另外，SimpleJdbcTemplate还声明了一部分模板方法，可以接受使用命名参数符号作为占位符的SQL语句及以Map或者SqlParameterSource的形式传入的参数。这部分模板方法会将执行逻辑委派给SimpleJdbcTemplate内部持有的NamedParameterJdbcTemplate实例，这也就是为什么会说，SimpleJdbcTemplate集JdbcTemplate和NamedParameterJdbcTemplate的功能于一身。可以通过SimpleJdbcTemplate.getNamedParameterJdbcOperations()方法获得NamedParameterJdbcTemplate的实例引用，而通过SimpleJdbcTemplate.getJdbcOperations()获得JdbcTemplate的实例引用，如下所示：

```
SimpleJdbcTemplate sjdbcTemplate = ...;
NamedParameterJdbcOperations npJdbcOper = sjdbcTemplate.getNamedParameterJdbcOperations();
JdbcOperations jdbcOper = sjdbcTemplate.getJdbcOperations();
```

从这里可以看出来，SimpleJdbcTemplate的最终工作还是委派给了JdbcTemplate，它本身只是在JdbcTemplate和NamedParameterJdbcTemplate基础上加了部分Java 5的功能修饰。

如果说NamedParameterJdbcTemplate是在JdbcTemplate的基础上套了一层网，那么，SimpleJdbcTemplate则是又在NamedParameterJdbcTemplate的基础上套了另一层网而已。

14.1.4 Spring中的DataSource

Spring的数据访问框架在数据库资源的管理上全部采用JDBC 2.0标准之后引入的javax.sql.DataSource接口作为标准，无论是JdbcTemplate还是各种ORM方案的集成，皆是如此。鉴于DataSource的重要地位，我们有必要对其做进一步的了解。

1. DataSource的种类

DataSource的基本角色是ConnectionFactory，所有的数据库连接将通过DataSource接口统一管理。DataSource实现类根据功能强弱可以划分为以下三类。

- 简单的DataSource实现

这种DataSource实现通常只提供作为ConnectionFactory角色的基本功能，更多时候，我们会使用这类DataSource实现进行开发或者测试，而绝不会用于正式的生产环境。Spring在org.springframework.jdbc.datasource包下提供了如下两个简单的DataSource实现。

org.springframework.jdbc.datasource.DriverManagerDataSource。顾名思义，DriverManagerDataSource的提出，主要是为了替换最古老的基于java.sql.DriverManager获取连接的方

式。如果你从最初的1.x版本就开始使用JDBC，那么应该还记得最初是如何进行数据库连接的（此处忽略了异常处理），如代码清单14-36所示。

代码清单14-36 使用DriverManager连接数据库的代码示例

```
// 1. 初始化Driver类
Class.forName("driverClassName",true,getClass().getClassLoader());
// 2. 由DriverManager取得连接
String jdbcUrl = ...;
Properties connectionInfo = new Properties();
connectionInfo.put("user","..");
connectionInfo.put("password","..");
...
Connetion con = DriverManager.getConnection(jdbcUrl, connectionInfo);
...
```

当每次向DriverManager请求一个数据库连接的时候，DriverManager都会返回一个新的数据库连接。实际上，DriverManagerDataSource就是对这种行为以DataSource标准接口进行封装后的产物。当每次通过DriverManagerDataSource的getConnection()方法请求连接的时候，DriverManagerDataSource也会每次返回一个新的数据库连接。也就是说，DriverManagerDataSource没有提供连接缓冲池的功能，在某些情况下应该避免将其应用于生产环境。

我们可以通过编程的方式或者通过Spring的IoC容器来使用DriverManagerDataSource，唯一要做的就是提供必要的连接信息，以下给出了通过Spring的IoC容器配置使用DriverManagerDataSource的代码示例：

```
<bean id="dataSource"➥
class="org.springframework.jdbc.datasource.DriverManagerDataSource" ➥
destroy-method="close">
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql://localhost/mysql?useUnicode=true&amp; ➥
    characterEncoding=utf8"/>
    <property name="username" value="..."/>
    <property name="password" value="..."/>
</bean>
```

destroy-method="close"的作用，就不用多说了吧？

org.springframework.jdbc.datasource.SingleConnectionDataSource。SingleConnectionDataSource是在DriverManagerDataSource的基础上构建的一个比较有意思的DataSource实现。在每次请求的时候，DriverManagerDataSource都返回新的数据库连接，而SingleConnectionDataSource则是每次都返回同一个数据库连接，也就是说，SingleConnectionDataSource只管理并返回一个数据库连接，或者可以说是，维护一个singleton的Connection的ConnectionFactory。

从SingleConnectionDataSource取得的Connection，在被调用方通过close()方法关闭之后，当再次从SingleConnectionDataSource请求数据库连接时，将抛出SQLException。所以，如果我们想即使从SingleConnectionDataSource返回的Connection对象的close()方法被调用，连接也不关闭，可以通过SingleConnectionDataSource的setSuppressClose(boolean)方法将SingleConnectionDataSource的suppressClose设置为true。此后，只要SingleConnectionDataSource不被销毁，那么从它那里返回的Connection对象将一直存在。 14

SingleConnectionDataSource提供了多个构造方法，可以像DriverManagerDataSource那样构造SingleConnectionDataSource，也可以对现有的数据库连接进行封装，比如：

```
Connection availableConnection = ...;
SingleConnectionDataSource dataSource = ➥
new SingleConnectionDataSource(availableConnection,true);
```

对于遗留系统和现有系统的集成，这种方式或许会有意想不到的帮助。

- 拥有连接缓冲池的DataSource实现

这一类DataSource实现，除了提供作为ConnectionFactory角色的基本功能之外，内部还会通过连接缓冲池对数据库连接进行管理，生产环境下的DataSource全都属于这一类。使用数据库连接缓冲池，可以在系统启动之初就初始化一定数量的数据库连接以备用，返回给客户端的Connection对象通过close()方法被关闭，实际上只是被返回给缓冲池，而不是真正的被关闭。这极大地促进了数据库连接资源的复用，进而提高系统性能。

Jakarta Commons DBCP和C3P0是这类DataSource实现中的代表（当然，WebLogic、WebSphere、JBoss等应用服务器提供的DataSource实现也属于这个范畴），我们可以在独立的应用程序（standalone application）中使用它们，而不用非要绑定到应用服务器。前面已经多次见过DBCP的使用，是通过编程方式使用，还是IoC容器配置方式使用，可以根据情况来定。下面我们来看看C3P0的配置（如下所示）。

```
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
    <property name="driverClass" value="com.mysql.jdbc.Driver" />
    <property name="jdbcUrl" value="${jdbc.url}" />
    <property name="user" value="${jdbc.username}" />
    <property name="password" value="${jdbc.password}" />
</bean>
```

ComboPooledDataSource是C3P0提供的DataSource实现，除了可以指定基本的连接信息，也可以指定初始连接数、最小连接数、最大连接数等参数。更多的信息可以参照C3P0的API文档。

- 支持分布式事务的DataSource实现类

确切地说，这一类的DataSource实现类应该是javax.sql.XADataSource的实现类，从XADataSource返回的数据库连接类型为javax.sql.XAConnection，而XAConnection扩展了javax.sql.PooledConnection，所以也可以看出来，支持分布式事务的DataSource实现类，同样支持数据库连接的缓冲。

除非应用程序确实需要分布式事务，否则，我们没有必要使用这一类的DataSource实现，通常的拥有连接缓冲池的DataSource实现类就足够了。而且，通常只有比较重量级的应用服务器才会提供支持分布式事务的DataSource，比如BEA的WebLogic或者IBM的WebSphere等。

纵观整个DataSource家族，DBCP和C3P0将是我们最常用的DataSource实现，无论开发、测试还有生产环境。最重要的是，我们可以在Spring容器中很容易的配置和使用它们。

2. DataSource的访问方式

我们可以在本地构造并持有相应的DataSource实现，也可以将相应的DataSource绑定到命名服务，这完全取决于当前应用程序的使用场景。不过，DataSource的位置将决定我们获取DataSource支持的不同方式。

- 本地DataSource访问

如果我们在当前应用程序的上下文中构造并持有了相应的DataSource实现，那么，可以通过本地的DataSource引用对其进行访问。这将是最常用的方式，无论是在独立的应用程序中还是在应用服务器中。

只要将相应的DataSource实现类所在的jar包加入应用程序的Classpath，我们就可以直接构造并访问它。而使用Spring的IoC容器来管理本地的DataSource资源将是最为理想的方式，只要在容器的配

置文件中进行简单的配置即可，容器中任何需要该资源的对象都可以获得依赖注入。否则，或许得自己构建一个专门用于该DataSource存取的Singleton对象，以供相应对象调用。

现在，我们对“如何使用Spring的IoC容器配置和访问DataSource”已经非常熟悉了，所以废话少说，接着看下一种DataSource的访问方式如何？

- 远程DataSource访问

对于各种应用服务器提供的特有的DataSource实现，或者绑定到应用服务器命名服务的独立的DataSource实现，我们需要通过JNDI对其进行访问。对于运行于应用服务器的程序或者分布式应用来说，通过JNDI访问DataSource将是最为常见的方式。

在Spring的IoC容器中，我们可以通过org.springframework.jndi.JndiObjectFactoryBean对这些DataSource进行访问，例如：

```
<bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
      <property name="jndiName">
          <value>java:env/myDataSource</value>
      </property>
</bean>
```

JndiObjectFactoryBean是一个FactoryBean实现，容器中对其引用获得的将是getObject()所返回的对象，而不是JndiObjectFactoryBean本身。在这里，当然就是我们要用的DataSource。

如果我们已经迁移到Spring 2.x，并且使用的是基于XSD的配置格式，那么我们可以直接使用<jndi:lookup>来查找应用服务器上的DataSource，如代码清单14-37所示。

代码清单14-37 通过<jndi:lookup>查找DataSource配置示例

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans" ➥
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ➥
xmlns:jee="http://www.springframework.org/schema/jee" ➥
xsi:schemaLocation="http://www.springframework.org/schema/beans ➥
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd ➥
http://www.springframework.org/schema/jee ➥
http://www.springframework.org/schema/jee/spring-jee-2.0.xsd">

     <jndi:lookup id="dataSource" jndi-name="java:env/myDataSource"/>
     ...
</beans>
```

相对于通过直接指定JndiObjectFactoryBean来进行JNDI查找而言，基于XSD的方式显然简洁多了，而且所达到的效果是相同的。只不过，在使用前，不要忘记将jee的命名空间加入配置文件。

3. 自定义DataSource实现

除了可以使用现有的各种DataSource实现，Spring在org.springframework.jdbc.datasource包下还提供了部分类帮助我们自定义实现相应的DataSource，前提是当前需求实在是比较特殊。

- 新建DataSource实现

要实现一个新的DataSource，我们可以扩展org.springframework.jdbc.datasource.AbstractDataSource，这是专门用于DataSource实现的扩展基类。它的一个直接实现类我们已经接触过了，就是DriverManagerDataSource，而SingleConnectionDataSource扩展了DriverManagerDataSource，也算AbstractDataSource一家子了。

在Spring框架中，AbstractDataSource除了DriverManagerDataSource这一个直接子类外，还有一个org.springframework.jdbc.datasource.lookup.AbstractRoutingDataSource实现类，这个类的作用比较特殊，但某些场景下应该很有用。AbstractRoutingDataSource会持有一组DataSource，当它的getConnection()被调用的时候，会根据条件从这组DataSource中查找符合条件的DataSource，然后调用查找取得的DataSource上的getConnection()。如果应用程序中有多个数据库，并且需要根据情况让应用程序访问不同的数据库，那么扩展并实现AbstractRouting-DataSource的一个子类将是我们的最佳选择。

要扩展AbstractRoutingDataSource，实际上只需要实现一个查找方法的逻辑即可，如下：

```
protected abstract Object determineCurrentLookupKey();
```

每次getConnection()请求将被导向哪个DataSource，将由这个方法的逻辑来决定。

AbstractRoutingDataSource的整个场景，如图14-5所示。

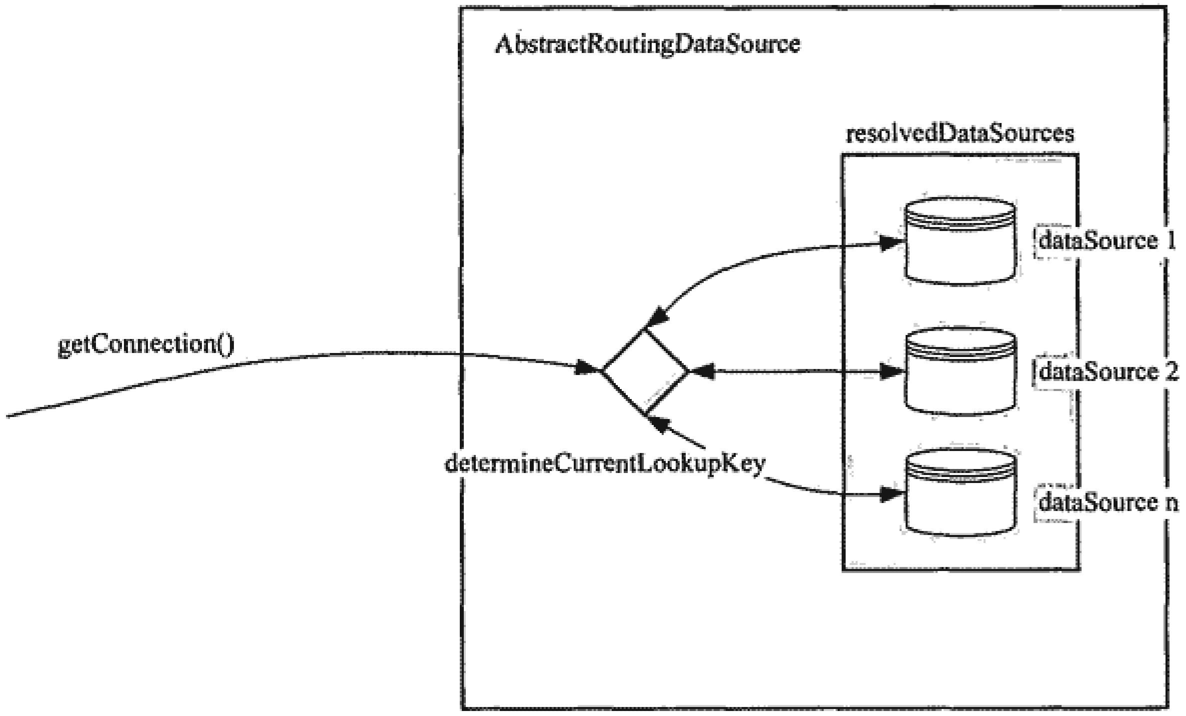

图14-5　AbstractRoutingDataSource请求处理场景

当然，到目前为止，都是Spring提供的AbstractDataSource扩展类。在我们的应用需求比较特殊，且找不到符合条件的DataSource的时候，那么也可以像DriverManagerDataSource或者Abstract-RoutingDataSource那样，直接扩展AbstractDataSource来实现我们自己的DataSource。不过，到目前为止我还没有遇到类似的需求场景，如果你遇到的话，那么，不要犹豫，动手做吧。

- 为现有DataSource添加新行为

要自定义DataSource实现，除了直接扩展org.springframework.jdbc.datasource.Abstract-DataSource或者它的相关子类之外，我们还有另一个选择，那就是org.springframework.jdbc.datasource.DelegatingDataSource。

DelegatingDataSource的作用是，它自身持有一个其他的DataSource实例作为目标对象，当getConnection()等方法被调用的时候，DelegatingDataSource会将调用转发（或者说委派）给持有的这个DataSource实例。DelegatingDataSource本身在转发这些调用之前没有做任何事情，这是

没有任何意义的，要发挥DelegatingDataSource的作用，我们在实现DelegatingDataSource子类的时候，需要覆写相应的方法，在转发方法调用之前添加相应的自定义逻辑。这听起来像是AOP的领域了，不过，目标对象和Joinpoint则是限定死的。

在Spring框架中，DelegatingDataSource有几个现成的实现类，如下所述。

org.springframework.jdbc.datasource.UserCredentialsDataSourceAdapter 。UserCredentialsDataSourceAdapter可以为现有的DataSource加入验证信息。通过UserCredentialsDataSourceAdapter的getConnection()方法获取Connection的时候，UserCredentialsDataSourceAdapter会根据自身的username和password信息，调用自身持有的DataSource目标对象的getConnection(username,password)方法以获取Connection。如果没有username和password的信息，再调用没有参数的getConnection()。

org.springframework.jdbc.datasource.TransactionAwareDataSourceProxy。TransactionAwareDataSourceProxy会对一个DataSource目标对象进行封装，所有从TransactionAwareDataSourceProxy取得的Connection将自动加入Spring的事务管理（事务管理相关内容将在下一部分内容中详述）。要想加入Spring的事务管理，我们需要使用DataSourceUtils类进行Connection的管理（JdbcTemplate内部就是这么做的），TransactionAwareDataSourceProxy内部在对DataSource目标对象的Connection管理上，也是使用DataSourceUtils进行的。所以，即使现在客户端使用JDBC API直接从TransactionAwareDataSourceProxy取得Connection，该Connection也将纳入Spring的事务管理，如下两种方式将产生相同的效果：

```
DataSource dataSource = ...;
Connection con = DataSourceUtils.getConnection(dataSource);
// 两种方式取得的Connection都可以加入Spring管理的事务
TransactionAwareDataSourceProxy txDataSource = ➥
new TransactionAwareDataSourceProxy(dataSource);
Connection txAwareConnection = txDataSource.getConnection();
```

更多TransactionAwareDataSourceProxy的信息可以参照Javadoc。

org.springframework.jdbc.datasource.LazyConnectionDataSourceProxy。通过LazyConnectionDataSourceProxy取得的Connection对象是一个代理对象，该代理对象可以保证当Connection被使用的时候才会从LazyConnectionDataSourceProxy持有的DataSource目标对象上获取。

我们可以根据情况选用这些现成的实现类，或者决定是否要根据需求去实现自己的。

14.1.5 JdbcDaoSupport

如果现在来实现一个DAO的话，肯定不会像原来那样使用底层的JDBC API来实现它了。最起码，我们会使用相应的DataSource提供数据库连接，使用JdbcTemplate进行数据库操作，基本上类似代码清单14-38所演示的内容。

14

代码清单14-38 使用JdbcTemplate和DataSource的DAO实现类原型代码示例

```
public class GenericDao implements IDaoInterface
{
    private DataSource   dataSource;
    private JdbcTemplate jdbcTemplate;
    public GenericDao(DataSource ds,JdbcTemplate jt)
    {
        this.dataSource   = ds;
        this.jdbcTemplate = jt;
```

```
    }
    public void update(DomainObject obj)
    {
        getJdbcTemplate().update(...);
    }
    ...

    // setter和getter方法定义
}
```

单个的DAO实现这么做是没有问题的，但当存在多个类似的DAO实现的时候，我们就会考虑重构这些DAO实现类，将DataSource和JdbcTemplate全部提取到统一的超类中。不过，这部分工作Spring已经为我们做了，它直接提供了org.springframework.jdbc.core.support.JdbcDaoSupport作为所有基于JDBC进行数据访问的DAO实现类的超类。所以，DAO直接继承JdbcDaoSupport就可以，如下所示：

```
public class GenericDao extends JdbcDaoSupport implements IDaoInterface
{
    public void update(DomainObject obj)
    {
        getJdbcTemplate().update(...);
    }
    ...

    // setter和getter方法定义
}
```

至此，基于JdbcTemplate方式进行数据访问的所有内容就告一段落了，我们要进入下一单元——基于操作对象的JDBC使用方式。

14.2　基于操作对象的 JDBC 使用方式

Spring除了提供基于Template形式的JDBC使用方式，还对各种数据库操作以面向对象的形式进行建模，为我们使用JDBC进行数据访问提供了另一种视角。

在这种基于操作对象的JDBC使用方式中，查询、更新、调用存储过程等数据访问操作，被抽象为操作对象，这些操作对象统一定义在org.springframework.jdbc.object包下，以org.springframework.jdbc.object.RdbmsOperation作为整个操作对象体系的顶层抽象定义（见图14-6）。

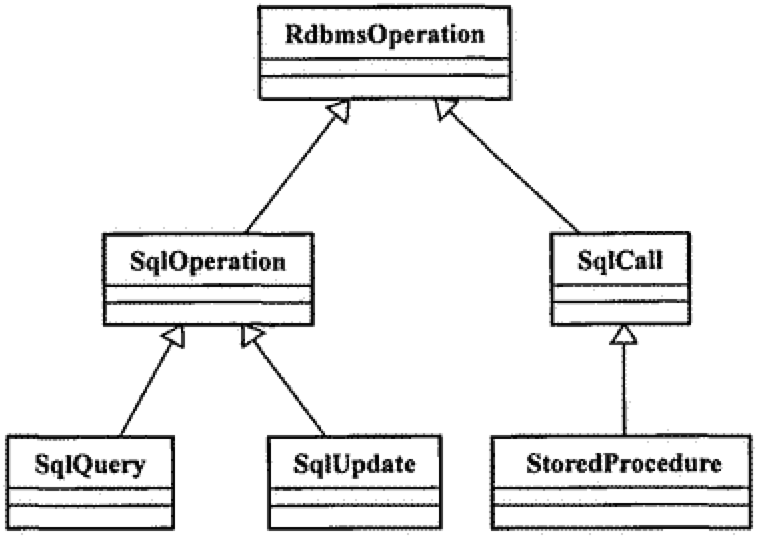

图14-6　操作对象的体系

RdbmsOperation是一个抽象类，它提供了所有子类所需要的公共设施，包括当前数据库操作对应的SQL语句的声明，参数列表处理，以及进行底层数据库操作所必需的JdbcTemplate实例等。所有的操作对象最终的数据访问都是通过JdbcTemplate进行的，这一点可以让我们清楚，实际上，基于操作对象的JDBC使用方式与基于JdbcTemplate的JDBC使用方式是统一的，只不过对待概念的视角上有所不同而已。

根据数据访问操作，RdbmsOperation分为三个主要分支，即查询操作对象分支、更新操作对象分支以及存储过程对象分支。

- **查询操作对象分支**。org.springframework.jdbc.object.SqlQuery抽象类是查询操作对象分支的“首脑”，它为子类提供了各种执行数据查询操作的方法实现，不过，查询结果的处理留给子类来实现。
- **更新操作对象分支**。org.springframework.jdbc.object.SqlUpdate是该分支的主要实现类，它是实体类，我们可以直接使用它进行数据库更新操作，也可以对其进行扩展，比如，在子类中提供基于强类型参数的更新方法定义。
- **存储过程对象分支**。在这个分支中，我们将更多地使用org.springframework.jdbc.object.StoredProcedure，而不是它的父类org.springframework.jdbc.object.SqlCall。因为SqlCall的主要作用是根据调用信息构建相应的CallableStatementCreator，StoredProcedure在SqlCall的基础上提供了执行调用存储过程的方法定义。

SqlQuery和SqlUpdate的父类是org.springframework.jdbc.object.SqlOperation，该类的主要工作是根据提供的SQL语句和参数列表信息，为子类SqlQuery和SqlUpdate提供相应的PreparedStatementCreator。这样，SqlQuery和SqlUpdate直接根据父类提供的PreparedStatementCreator进行数据的查询和更新即可。

现在，让我们详细看一下每一个分支下，Spring具体提供了哪些可用的操作对象。

14.2.1 基于操作对象的查询

图14-7给出的是整个查询操作对象体系。

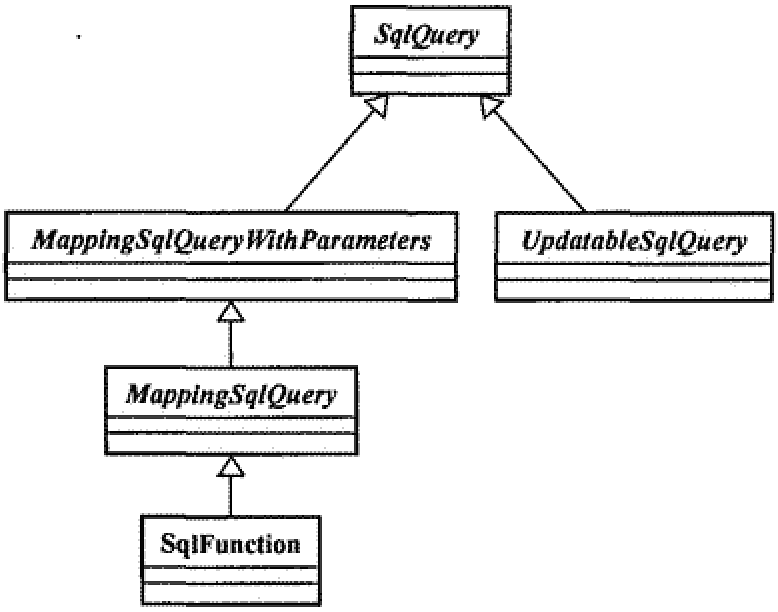

图14-7 查询操作对象体系

14

相对于其他操作对象定义分支来说，查询操作对象的定义体系看起来要“繁盛”得多。

1. **MappingSqlQueryWithParameters**

SqlQuery作为MappingSqlQueryWithParameters的父类，只定义了执行数据查询操作的各种方法，而处理查询结果的工作则留给了具体子类，相应子类需要实现SqlQuery定义的如下抽象方法，以返回处理查询结果的RowMapper对象：

```
protected abstract RowMapper newRowMapper(Object[] parameters, Map context);
```

MappingSqlQueryWithParameters对象提供了该方法的实现，返回了MappingSqlQueryWithParameters的内部类RowMapperImpl的实例，作为处理查询结果所需要的RowMapper，RowMapperImpl的定义见代码清单14-39。

代码清单14-39　RowMapperImpl定义

```
protected class RowMapperImpl implements RowMapper {

    private final Object[] params;

    private final Map context;

    /**
     * Use an array results. More efficient if we know how many results to expect.
     */
    public RowMapperImpl(Object[] parameters, Map context) {
        this.params = parameters;
        this.context = context;
    }

    public Object mapRow(ResultSet rs, int rowNum) throws SQLException {
        return MappingSqlQueryWithParameters.this.mapRow(rs, rowNum, this.params, this. ➥
        context);
    }
}
```

不过，RowMapperImpl在实现RowMapper的mapRow方法的时候，将具体的处理逻辑转发给了MappingSqlQueryWithParameters的mapRow方法，如下所示：

```
protected abstract Object mapRow(ResultSet rs, int rowNum, Object[] parameters, Map context) ➥
throws SQLException;
```

所以，要使用MappingSqlQueryWithParameters进行数据查询，我们得提供它的子类，然后实现这个mapRow方法以处理查询结果。

还记得我们前面的fx_news表吗？现在我们要使用MappingSqlQueryWithParameters对其进行查询。首先，得定义MappingSqlQueryWithParameters的一个子类，为mapRow方法提供合适的逻辑以处理查询结果。代码清单14-40给出了针对fx_news表查询用的MappingSqlQueryWithParameters实现子类的定义。

代码清单14-40　MappingSqlQueryWithParameters实现子类定义代码示例

```
public class FXNewsQueryWithParameters extends MappingSqlQueryWithParameters {

    private static final String QUERY_SQL = "select * from fx_news where news_title = ?";

    public FXNewsQueryWithParameters(DataSource dataSource)
    {
```

```
        super(dataSource, QUERY_SQL);
        declareParameter(new SqlParameter(java.sql.Types.VARCHAR));
        compile();
    }

    @Override
    protected Object mapRow(ResultSet rs, int row, Object[] parameters, Map context) ➥
    throws SQLException {
        FXNewsBean newsBean = new FXNewsBean();
        newsBean.setNewsId(rs.getString(1));
        newsBean.setNewsTitle(rs.getString(2));
        newsBean.setNewsBody(rs.getString(3));
        return newsBean;
    }
}
```

关于FXNewsQueryWithParameters的定义，我们有几点需要说明，通常，这些点将同样应用于其他操作对象，如下所述。

- 所有的操作对象都需要一个DataSource用于访问数据库。底层使用JdbcTemplate进行最终的数据访问操作的事实，让操作对象需要一个DataSource变得很自然。
- 在使用操作对象（RdbmsOperation）之前，需要调用compile()方法对操作对象所必需的各种参数进行检查和设置，只有compile()通过之后，当前操作对象才可以使用。所以，我们在FXNewsQueryWithParameters的构造方法最后调用了compile()方法，这样FXNewsQueryWithParameters一旦构造完成即可付诸使用。
- declareParameter()方法用于为操作对象指明所对应的SQL语句中各种参数类型，该方法接受org.springframework.jdbc.core.SqlParameter及其子类作为参数。因为SQL中有news_title这一参数，所以，我们使用declareParameter()告知操作对象这一参数的相关信息。

  注意，declareParameter()必须在compile()之前进行。
- 使用mapRow方法与结合使用JdbcTemplate和RowMapper处理查询结果没有太大区别，唯一需要关注的是最后两个方法参数。不过，在此之前，我们先来看一下如何使用已经定义完的FXNewsQueryWithParameters。

要使用FXNewsQueryWithParameters，我们只需要为它提供一个DataSource，剩下的就是取得查询结果进行处理，如下所示：

```
DataSource dataSource = ...; // DBCP 或者 C3P0或者其他数据源实现
FXNewsQueryWithParameters query = new FXNewsQueryWithParameters(dataSource);
List news = query.execute("FX Market Rocks");
for(int i=0,size=news.size();i<size;i++)
{
    FXNewsBean bean = (FXNewsBean)news.get(i);
    System.out.println(bean);
}
```

如果多个DAO都需要用到FXNewsQueryWithParameters所提供的查询逻辑的话，我们可以将同一个FXNewsQueryWithParameters实例，分别注入这些需要的对象中，compile()后的FXNewsQueryWithParameters是线程安全的。

有关查询操作对象所定义更多的查询方法定义，可以参照Spring的Javadoc文档。现在，我们要回头看一下MappingSqlQueryWithParameters的mapRow方法的最后两个参数，如下。

Object[] parameters。调用execute方法或者其他查询方法所传入的参数值列表。比如，我们在上面调用execute方法的时候传入了一个参数值“FX Market Rocks”，那么，就可以通过parameters获取这个值。我们在mapRow中查看parameters[0]，对应的应该就是“FX Market Rocks”。不过，处理查询结果的时候，好像很少要用到SQL对应的参数值吧？

Map context。对于查询操作对象所定义的查询方法来说，每组查询方法通常对应一个可以传入Map作为参数的重载方法（如下所示）：

```
List   execute(Object[] params)
List   execute(Object[] params, Map context)
...
Object findObject(String p1)
Object findObject(String p1, Map context)
...
```

该参数的作用在于，可以为查询结果的处理传入更多信息。我们可以在mapRow方法中使用这些传入的信息对查询结果进行某些处理。

比如，在查询获得新闻信息的基础上，我们要在新闻标题上增加某个字符串前缀，然后再返回，就可以通过这个Map参数传入所需要的前缀信息，代码清单14-41演示了这种逻辑的实现。

代码清单14-41　包含context参数的查询方法使用示例

```
FXNewsQueryWithParameters query = new FXNewsQueryWithParameters(dataSource);

Map context = new HashMap();
context.put("PREFIX", "FX");

List news = query.execute("title",context);
for(int i=0,size=news.size();i<size;i++)
{
    FXNewsBean bean = (FXNewsBean)news.get(i);
    System.out.println(bean);
}
```

我们现在使用的execute接收参数值和传入的context信息，那么context传入的信息，就可以通过FXNewsQueryWithParameters的mapRow方法最后一个参数取得。代码清单14-42演示了context信息的使用。

代码清单14-42　传入的context信息的使用代码示例

```
@Override
protected Object mapRow(ResultSet rs, int row, Object[] parameters, Map context) ➥
throws SQLException {
    String prefix = (String)context.get("PREFIX");

    FXNewsBean newsBean = new FXNewsBean();
    newsBean.setNewsId(rs.getString(1));
    newsBean.setNewsTitle(prefix+rs.getString(2));
    newsBean.setNewsBody(rs.getString(3));
    return newsBean;
}
```

当然，context传入的信息并不限于简单的字符串前缀，这要根据当前的应用场景来决定。

可以看出，mapRow方法的最后两个参数在日常开发中并非经常用到，因此，Spring框架给出了MappingSqlQueryWithParameters的另一个子类，即MappingSqlQuery。

2. MappingSqlQuery

MappingSqlQuery继承了MappingSqlQueryWithParameters，然后实现了MappingSqlQueryWithParameters拥有4个参数的mapRow方法，实现的方法逻辑如下所示：

```
protected final Object mapRow(ResultSet rs, int rowNum, Object[] parameters, Map context) ➥
throws SQLException {
    return mapRow(rs, rowNum);
}
```

MappingSqlQuery对该方法的实现，实际上只是去掉了最后的两个方法参数，然后又重新为子类公开了只有前两个方法参数的mapRow方法，如下：

```
protected abstract Object mapRow(ResultSet rs, int rowNum) throws SQLException;
```

这样，要通过扩展MappingSqlQuery进行数据查询的话，只需要实现拥有两个参数的mapRow方法就可以。

定义同样的查询，使用MappingSqlQuery进行查询对象定义的话，看起来如代码清单14-43所示。

代码清单14-43 MappingSqlQuery定义和使用代码示例

```
public class FXNewsQuery extends MappingSqlQuery {

    private static final String QUERY_SQL = "select * from fx_news where news_title = ?";

    public FXNewsQuery(DataSource dataSource)
    {
        super(dataSource, QUERY_SQL);
        declareParameter(new SqlParameter(Types.VARCHAR));
        compile();
    }

    @Override
    protected Object mapRow(ResultSet rs, int row) throws SQLException {
        FXNewsBean newsBean = new FXNewsBean();
        newsBean.setNewsId(rs.getString(1));
        newsBean.setNewsTitle(rs.getString(2));
        newsBean.setNewsBody(rs.getString(3));
        return newsBean;
    }
}
```

除了要实现的mapRow方法不同，与直接继承MappingSqlQueryWithParameters并无二致。至于怎么使用这个FXNewsQuery，与FXNewsQueryWithParameters的使用简直是一模一样的。

通过继承MappingSqlQuery实现查询操作对象，与结合RowMapper使用JdbcTemplate实际上应该是对等的。所以，如果在结果集的处理上没有其他附加条件的话，MappingSqlQuery将是我们最经常使用的查询操作对象实现了。

14

3. SqlFunction

SqlFunction是职责更加专一的查询操作对象，它负责的是只返回一行并且一列的查询结果的查询。SqlFunction本身的目的就是为了调用只返回单一查询结果的简单查询，比如select count(*) from fx_news或者select sysdate from dual。

其他的SqlQuery子类都是抽象类，SqlFunction本身则为实体类，我们可以直接实例化SqlFunction并调用相应的查询方法。代码清单14-44简单地演示了SqlFunction的使用。

代码清单14-44 SqlFunction使用示例

```
DataSource dataSource = ...; // DBCP或者C3P0或者其他数据源实现
SqlFunction function = new SqlFunction(dataSource,"select count(*) from fx_news");
function.compile();
int count = function.run();
// 或者
SqlFunction f = new SqlFunction(dataSource,"select current_timestamp() from fx_news limit 1");
f.compile();
Object currentTime = f.runGeneric();
assertTrue(currentTime instanceof java.sql.Timestamp);
```

SqlFunction的run()方法返回int型结果，runGeneric()方法返回Object型结果，二者都可以接收参数。

因为SqlFunction同样属于RdbmsOperation，所以使用之前，不要忘记调用它的compile()方法。

4. UpdatableSqlQuery

UpdatableSqlQuery主要对应可更新结果集的查询，通过它，我们可以对查询后的结果进行更新操作。通常情况下，如果数据库中的某些数据需要统一的更新，我们可以将这些数据查询出来，然后再分批更新回数据库。不过，既然我们的主要目的是更新，而查询的结果是次要的，那么可以在可更新结果集上进行类似操作，而UpdatableSqlQuery就是为此而生。

要使用UpdatableSqlQuery，只需要继承它，然后实现updateRow方法即可（方法定义如下）：

```
protected abstract Object updateRow(ResultSet rs, int rowNum, Map context) throws ➥
SQLException;
```

假设我们要将fx_news表中所有新闻标题的首字母全部变为大写，那么，我们可以有类似代码清单14-45的UpdatableSqlQuery实现类。

代码清单14-45 CapitalTitleUpdateableSqlQuery类定义代码

```
public class CapitalTitleUpdateableSqlQuery extends UpdatableSqlQuery {

    public CapitalTitleUpdateableSqlQuery(DataSource dataSource,String sql)
    {
        super(dataSource,sql);
        compile();
    }

    @Override
    protected Object updateRow(ResultSet rs, int row, Map context) ➥
    throws SQLException {
        String title = rs.getString("news_title");
        rs.updateString("news_title", StringUtils.capitalize(title));
        return null;
    }
}
```

要使用该CapitalTitleUpdateableSqlQuery实现类，与使用其他的SqlQuery类似，只不过，现在的查询结果对于我们来说已经没有太大用处了（使用见如下示例代码）。

```
DataSource dataSource = ...; // DBCP或者C3P0或者其他数据源实现
String sql = "select * from fx_news";
CapitalTitleUpdateableSqlQuery updatableQuery = new CapitalTitleUpdateableSqlQuery ➥
(dataSource,sql);
updatableQuery.execute();
```

可能你也注意到了，updateRow方法最后一个参数为Map context，它的作用与我们在讲解MappingSqlQueryWithParameters的mapRow方法的时候所提到的最后一个参数是一样的。在我们使用可更新结果集对查询结果进行更新的过程中，如果需要某些外部信息的话，我们可以在调用UpdatableSqlQuery数据访问操作方法的时候通过Map传入，而在updateRow方法中通过Map context参数取得，使用方式可以参照MappingSqlQueryWithParameters部分（14.2.1节的第1小节）针对该参数的使用示例。

5. 基于操作对象的LOB查询

在讲解JdbcTemplate的时候，我们讲解了如何使用JdbcTemplate对LOB型数据进行查询。现在，我们要说的是使用查询操作对象进行LOB型数据的查询。

实际上，不管是使用JdbcTemplate还是使用查询操作对象，LOB数据都是由具体的LobHandler来处理的。要在查询操作对象中对LOB型的查询结果进行处理，只要为具体的查询操作对象提供相应的LobHandler就行了。

同样是对images表进行查询，使用LobHandler处理LOB数据的查询操作对象如代码清单14-46所示。

代码清单14-46 用于LOB数据查询的操作对象定义示例

```
public class LobHandlingSqlQuery extends MappingSqlQuery {

    private static final String SQL = "select * from images where id=?";

    private LobHandler lobHandler = new DefaultLobHandler();

    public LobHandlingSqlQuery(DataSource dataSource)
    {
        setDataSource(dataSource);
        setSql(SQL);
        declareParameter(new SqlParameter(Types.INTEGER));
        compile();
    }

    @Override
    protected Object mapRow(ResultSet rs, int row) throws SQLException {
        Image image = new Image();
        image.setId(rs.getInt(1));
        image.setFilename(rs.getString(2));
        image.setEntity(lobHandler.getBlobAsBytes(rs, 3));
        image.setDescription(rs.getString(4));
        return image;
    }

    public LobHandler getLobHandler() {
        return lobHandler;
    }

    public void setLobHandler(LobHandler lobHandler) {
        this.lobHandler = lobHandler;
    }
}
```

当前的查询操作对象LobHandlingSqlQuery扩展了MappingSqlQuery，根据具体应用情况，我们同样可以扩展其他SqlQuery类。在LobHandlingSqlQuery中，我们使用LobHandler将BLOB型的图

像文件数据以byte[]的形式取得，供查询调用方处理。现在，对LobHandlingSqlQuery查询获取的BLOB数据进行处理，如下所示：

```
LobHandlingSqlQuery lobQuery = new LobHandlingSqlQuery(dataSource);
List images = lobQuery.execute(1);
Image image = (Image)images.get(0);
FileUtils.writeByteArrayToFile(new File(image.getFilename()), image.getEntity());
```

在这里，我们是将从数据库中取得的转换为byte[]的BLOB数据写入了文件系统。

14.2.2 基于操作对象的更新

Spring提供的用于更新的操作对象主要有两个，即SqlUpdate和BatchSqlUpdate，前者主要用于基本的更新操作，后者则主要用于进行批量更新操作。

下面让我们详细看一下这两种更新操作对象。

1. SqlUpdate

SqlUpdate与SqlQuery同样继承自SqlOperation，所以它的所有的更新操作全部基于SqlOperation父类所返回的PreparedStatementCreator进行。

SqlUpdate是实体类，我们可以直接使用它进行数据更新，而不用必须继承它然后通过子类来使用，如下代码演示了该类的使用：

```
String updateSql = "update fx_news set news_title=? where news_id=?";
SqlUpdate sqlUpdate = new SqlUpdate(dataSource,updateSql);
sqlUpdate.declareParameter(new SqlParameter(Types.VARCHAR));
sqlUpdate.declareParameter(new SqlParameter(Types.INTEGER));
sqlUpdate.compile();

int affectedRows = sqlUpdate.update(new Object[]{"New Title",new Integer(1)});
```

直接使用SqlUpdate需要注意的就是，每次我们都需要根据情况，通过declareParameter声明相应的SQL占位符参数，并且最后，不要忘了最主要的，一定要在调用compile方法之后，才可以调用相应的update方法进行数据更新操作。

除了直接使用SqlUpdate进行数据更新，我们也可以通过继承的方式来使用它，这样我们可以得到的好处有如下几点。

- 封装具体的更新操作，将初始化SqlUpdate的各个步骤统一到一处管理，避免遗漏（通常，declareParameter和compile方法在直接使用的时候难免出现遗漏的情况）。
- 提供强类型参数的更新操作方法，以替代弱类型参数的更新方法，进一步强化调用方与被调用方的契约关系。

如果我们对刚才的更新操作进行封装的话，就可以有如代码清单14-47所示的更新操作对象定义。

代码清单14-47 子类化的SqlUpdate对象定义代码示例

```
public class NewsTitleSqlUpdate extends SqlUpdate {

    private static final String TITLE_UPDATE_SQL = "update fx_news set news_title=? ➥
    where news_id=?";

    public NewsTitleSqlUpdate(DataSource dataSource)
    {
        super(dataSource,TITLE_UPDATE_SQL);
        declareParameter(new SqlParameter(Types.VARCHAR));
        declareParameter(new SqlParameter(Types.INTEGER));
```

```
        compile();
    }

    public int updateTitleById(String newTitle,int id)
    {
        return update(new Object[]{newTitle,new Integer(id)});
    }
}
```

而现在对于调用方来说，使用NewsTitleSqlUpdate进行数据更新要比直接使用SqlUpdate简洁得多，如下所示：

```
NewsTitleSqlUpdate update = new NewsTitleSqlUpdate(dataSource);
int affectedRows = update.updateTitleById("new title", 1);
```

是直接使用SqlUpdate还是子类化后再使用，这完全可以根据应用的具体情况来决定。如果某个更新操作的可重用性很强，那么针对该更新操作，子类化SqlUpdate后再使用比较合适。而如果某个更新操作在很少的场景用到，那么直接使用SqlUpdate则比较合适。

2. BatchSqlUpdate

在Spring中，除了使用JdbcTemplate进行数据的批量更新之外，BatchSqlUpdate为我们提供了另一种选择。BatchSqlUpdate底层也是JdbcTemplate结合BatchPreparedStatementSetter进行批量更新，没有更多的奥秘，所以，我们还是直接来看BatchSqlUpdate的使用吧！

BatchSqlUpdate同SqlUpdate一样也是实体类，所以，我们可以直接使用它。如果我们要将获得的新闻条目批量更新到数据库，那么可以使用代码清单14-48所示的代码进行。

代码清单14-48 BatchSqlUpdate使用代码示例

```
public void batchInsertFXNews(List<FXNewsBean> newsList)
{
    BatchSqlUpdate batchUpdate = new BatchSqlUpdate(dataSource);
    batchUpdate.setSql("INSERT INTO fx_news(news_id, news_title, news_body) ➥
    VALUES(?, ?, ?)");
    batchUpdate.declareParameter(new SqlParameter(Types.BIGINT));
    batchUpdate.declareParameter(new SqlParameter(Types.VARCHAR));
    batchUpdate.declareParameter(new SqlParameter(Types.LONGVARCHAR));
    batchUpdate.compile();

    for(FXNewsBean bean : newsList)
    {
        batchUpdate.update(new Object[]{bean.getNewsId(),bean.getNewsTitle(),bean. ➥
        getNewsBody()});
    }

    batchUpdate.flush();
}
```

14

我们使用BatchSqlUpdate的update方法将数据添加到批量更新的队列中，当队列数量等于batchSize的时候，将会自动触发批量更新操作，否则，数据只是被添加到更新队列而已，所以，最后需要调用flush()方法以确保所有的记录都更新到数据库。从这个角度来看，BatchSqlUpdate就像货车一样，货物装满即发车（数据量达到batchSize，就进行批量更新），当最后就剩下一点儿货物的时候，因为没有再多的货物可装载了，同样也得发这最后一趟，flush()方法就是这最后一趟车，确保即使数据量没有达到规定的batchSize，剩下的数据也得更新到数据库中。

我们可以通过setBatchSize(int)方法对batchSize进行调整，默认为5000，只有当默认值不足

以满足应用的批量更新性能要求的时候，才有必要对该数值进行调优。另外，与SqlUpdate的使用一样，如果感觉直接使用BatchSqlUpdate过于烦琐的话，我们也可以通过扩展它对其进行封装。

3. 基于操作对象的LOB更新

借助于LobHandler，我们同样可以使用更新操作对象对LOB型数据进行更新操作。

假设数据库中存放在images表的数据，对应某个文件名的图片文件需要替换，我们就可以提供一个支持LOB数据更新的SqlUpdate，对旧有的图片数据进行更新替换（见代码清单14-49）。

代码清单14-49　LOB数据的更新代码示例

```
public class ImageLobDataUpdate extends SqlUpdate {

    private static final String SQL = "update images set entity=? where filename=?";

    private LobHandler lobHandler = new DefaultLobHandler();

    public ImageLobDataUpdate(DataSource dataSource)
    {
        super(dataSource, SQL);
        declareParameter(new SqlParameter(Types.BLOB));
        declareParameter(new SqlParameter(Types.VARCHAR));
        compile();
    }

    public int replaceImageData(String filename,File imageFile) throws IOException
    {
        InputStream ins = null;
        try
        {
            ins = new FileInputStream(imageFile);
            SqlLobValue lobValue = new SqlLobValue(ins,(int)imageFile.length(),➥
            getLobHandler());
            return update(new Object[]{lobValue,filename});
        }
        finally
        {
            IOUtils.closeQuietly(ins);
        }
    }

    public LobHandler getLobHandler() {
        return lobHandler;
    }

    public void setLobHandler(LobHandler lobHandler) {
        this.lobHandler = lobHandler;
    }
}
```

关于ImageLobDataUpdate有如下几点需要说明。

- 要对LOB数据进行处理，我们需要一个LobHandler，所以，我们开始就声明了一个LobHandler。默认情况下是采用DefaultLobHandler，但如果数据库是Oracle，那么可以通过setLobHandler替换掉默认的LobHandler。
- 因为我们要更新的entity字段是BLOB型，所以，我们在使用declareParameter声明SQL中相应占位符参数的类型的时候，采用的是java.sql.Types.BLOB；

```
declareParameter(new SqlParameter(Types.BLOB));
```

其他的LOB类型处理需要根据具体应用场景进行调整。

- 在更新操作对象中，要为LOB型占位符参数提供参数值，需要使用org.springframework.jdbc.core.support.SqlLobValue，SqlLobValue将使用给定的LobHandler对具体的LOB数据进行处理。SqlLobValue的Javadoc中同样有结合SqlLobValue和JdbcTemplae进行LOB数据更新的示例。

有了ImageLobDataUpdate，剩下我们做的就是提供要替换的图片文件，指定要对数据库中哪个文件名对应的数据进行更新替换，如下所示：

```
ImageLobDataUpdate update = new ImageLobDataUpdate(dataSource);
update.replaceImageData("snow_image.jpg", new File("43.jpg"));
```

就是这么简单！当然，不要忘记异常的处理哦！

14.2.3 基于操作对象的存储过程调用

在讲解JdbcTemplate的时候，我们介绍过如何使用JdbcTemplate和相应的Callback接口来调用存储过程。现在，我们可以将JdbcTemplate和那些Callback接口放在脑后，因为专门针对存储过程调用的操作对象为我们带来了更便捷的方式。

StoredProcedure

org.springframework.jdbc.object.StoredProcedure是对应存储过程调用的操作对象，它通过其父类org.springframework.jdbc.object.SqlCall获得相应的底层API支持（CallableStatementCreator），然后在此基础上构建了调用存储过程的执行方法。

StoredProcedure是抽象类，所以需要实现相应子类，以封装对特定存储过程的调用。还记得我们在讲解JdbcTemplate调用存储过程的时候定义的存储过程吗？如下所示：

```
CREATE PROCEDURE CountTable(IN tableName varchar(1000),OUT sqlStr varchar(1000) , ➥
INOUT v INT)
BEGIN
    set @flag = v;
    set @sql = CONCAT('select count(*) into @res from ' , tableName , ' where ACTIVE_FLAG=?');
    PREPARE stmt FROM @sql;
    EXECUTE stmt using @flag;
    DEALLOCATE PREPARE stmt;
    set v = @res;
    set sqlStr = @sql;
END
```

通过继承StoredProcedure，我们可以为该存储过程的调用提供一个对应的操作对象（见代码清单14-50）。

14

代码清单14-50　StoredProcedure子类定义代码示例

```
public class CountTableStoredProcedure extends StoredProcedure {

    private static final String PROCEDURE_NAME = "CountTable";

    public static final String IN_PARAMETER_NAME = "tableName";
    public static final String OUT_PARAMETER_NAME = "sqlStr";
    public static final String INOUT_PARAMETER_NAME = "v";

    public CountTableStoredProcedure(DataSource dataSource)
```

```
    {
        super(dataSource,PROCEDURE_NAME);
        // setFunction(true);
        declareParameter(new SqlParameter(IN_PARAMETER_NAME,Types.VARCHAR));
        declareParameter(new SqlOutParameter(OUT_PARAMETER_NAME,Types.VARCHAR));
        declareParameter(new SqlInOutParameter(INOUT_PARAMETER_NAME,Types.INTEGER));
        compile();
    }

    public CountTableResult doCountTable(String tableName,Integer v)
    {
        Map paraMap = new HashMap();
        paraMap.put(IN_PARAMETER_NAME, tableName);
        paraMap.put(INOUT_PARAMETER_NAME, v);

        Map resultMap = execute(paraMap);

        CountTableResult result = new CountTableResult();
        result.setSql((String)resultMap.get(OUT_PARAMETER_NAME));
        result.setCount((Integer)resultMap.get(INOUT_PARAMETER_NAME));
        return result;
    }
}
```

该存储过程操作对象的部分细节我们有必要关注一下。

(1) 存储过程操作对象对应的SQL是存储过程的名称，而不是真正意义上的SQL语句。当调用compile方法的时候，StoredProcedure的父类SqlCall会根据提供的存储过程名称，拼装真正意义上的符合SQL92标准的存储过程调用语句，类似于{ call CountTable(?,?,?) }的形式。

因为我们的CountTableStoredProcedure只针对CountTable存储过程调用，所以，该存储过程的名称在类一开始就声明为常量：

```
private static final String PROCEDURE_NAME = "CountTable";
```

如果有多个存储过程的参数顺序相同，结果处理也一样的话，我们也可以将存储过程的名称声明为变量，这完全要取决于具体的应用场景。

(2) 在构造方法中，我们将setFunction(true);注释掉了，因为我们调用的CountTable不是一个Function。如果要调用的存储过程类型为Function的话，我们需要通过该方法将function的值设置为true，以告知StoredProcedure在处理调用的时候区别对待。

(3) 在complie之前通过declareParameter声明参数，这几乎是雷打不动的惯例。不过，在StoredProcedure中使用declareParameter的时候却要有所注意了：

- 随着存储过程参数类型为IN、OUT和INOUT的类型差异，declareParameter接受的参数类型也应该对应SqlParameter、SqlOutParameter和SqlInOutParameter；
- SqlParameter、SqlOutParameter和SqlInOutParameter的相应实例在构造的时候，必须指定对应的参数名称，因为在调用存储过程的时候，需要根据名称传入参数，更需要根据名称取得调用结果。

(4) StoredProcedure提供了execute方法执行存储过程调用，允许通过Map的形式传入调用所需要的IN或者INOUT类型的参数值，所以，在构建参数Map的时候，该Map中的键应该与调用declareParameter方法声明的参数名称相同。另外，execute执行后返回的结果也是Map形式，从该结果Map中取得具体的结果值的时候，也是通过declareParameter中声明的OUT/INOUT参数名作为键来获取的，所以，这就是我们将各个参数的名称在类定义的开始声明为常量的原因：

```
public static final String IN_PARAMETER_NAME = "tableName";
public static final String OUT_PARAMETER_NAME = "sqlStr";
public static final String INOUT_PARAMETER_NAME = "v";
```

无论是输入参数Map还是输出参数结果对应的Map，它们中的键应该与通过declareParameter方法声明的参数名称一一对应。

通过扩展StoredProcedure，不但封装了参数的声明和结果的提取，还为调用方提供了强类型的调用方法。现在，调用方可以通过doCountTable方法的强类型参数声明传入参数值，并取得强类型的CountTableResult对象作为结果，而不是泛泛的一个Map。

对于存储过程的调用者来说，它们的代码现在可以简洁到两行代码，如下所示：

```
// DataSource dataSource = ...;
CountTableStoredProcedure storedProcedure = new CountTableStoredProcedure(dataSource);
CountTableResult result = storedProcedure.doCountTable("tableName",1);
...
```

漂亮多了，不是吗？

StoredProcedure提供了两个execute方法执行存储过程的调用，一个就是我们刚才使用的通过Map提供输入参数的execute方法，另一个则是使用ParameterMapper类型提供输入参数的execute方法。那么，为什么要提供这个使用ParameterMapper类型提供输入参数的execute方法呢？

ParameterMapper定义的callback方法公开了相应的Connection。如果说在构造输入参数列表的时候，必须用到Connection的话，ParameterMapper恰好可以提供支持。比如，Oracle中定义的存储过程，接收数组类型作为参数，而在Oracle中，只能通过oracle.sql.ARRAY和相应的oracle.sql.ArrayDescriptor来定义数组类型的参数，ARRAY和ArrayDescriptor都需要用到相应的Connection进行构造。所以，对于Oracle中需要使用数组传入参数的存储过程来说，我们可以通过与代码清单14-51类似的代码进行调用。

代码清单14-51 ParameterMapper使用代码示例

```
public class OracleStoredProcedure
{
    ...

    public Map call(...)
    {
        ParameterMapper paramMapper = new ParameterMapper(){
            public Map createMap(Connection connection) throws SQLException {
                Map inMap = new HashMap();
                ...
                Integer[] params = ➥
                new Integer[]{new Integer(1),new Integer(2)};
                ArrayDescriptor desc = ➥
                new ArrayDescriptor("numbers", connection);
                ARRAY nums = new ARRAY(desc, connection, params);
                inMap.put("ArrayParameterName", nums);
                ...
                return inMap;
            }};

        return execute(paramMapper);
    }
}
```

14

当然，CountTableStoredProcedure在调用存储过程的时候也可以使用ParameterMapper传入相应的调用参数。只不过，ParameterMapper的createMap方法公开的Connection对于我们来说没有太大用。

> **注意**　有关通过StoredProcedure调用存储过程的更多信息，尤其是对Oracle数据库的存储过程的调用，可以参考*Professional Java Development with the Spring Framework*一书，其中提到了StoredProcedure的更多特性，另外，Spring的support论坛中也有更多相关内容。

14.3　小结

JDBC在Java平台上的数据访问领域一直占据重要地位，但在日常的开发过程中，糟糕的实践方式让JDBC的使用多少有些声名狼藉。Spring框架针对JDBC API在日常使用中暴露出来的种种问题，提供了一套有针对性的最佳实践方式，包括基于JdbcTemplate和基于操作对象的实践方式。

本章讲述了Spring框架提供的两种JDBC API的最佳实践方式，并详尽阐述了这两种最佳实践方式的实现原理和使用方式。

不过，要访问数据，JDBC并非唯一的选择。在ORM如此盛行的年代，不提及Java平台上强盛的ORM家族，或许有些说不过去。接下来，我们就一起来看一下Spring框架是如何为现有的各种ORM解决方案提供封装和集成的。

第 15 章

Spring对各种ORM的集成

本章内容

- Spring对Hibernate的集成
- Spring对iBATIS的集成
- Spring中对其他ORM方案的集成概述

Spring对当前各种流行的ORM解决方案的集成主要体现在以下几个方面。

- **统一的资源管理方式。**因为在使用各种ORM过程中，都会遇到与JDBC相似的资源管理问题，所以Spring框架以一致的方式，封装了各种使用ORM的功能，从而确保在整个Spring框架中，不管是使用JDBC还是使用ORM，它们的使用方式以及资源管理方式都是统一的。
- **特定于ORM的数据访问异常到Spring统一异常体系的转译。**为了能够让客户端只关注这个异常体系，而不用过多关注当前应用使用的是何种特定的数据访问技术，Spring在集成各种ORM解决方案的时候，自然要转译它们特定的数据访问异常，并纳入Spring统一的异常层次体系之下。
- **统一的数据访问事务管理及控制方式**[①]。Spring为各种数据访问方式（不只是ORM）提供的另一种统一的"公共设施"，就是它的事务管理抽象层。通过这个事务管理抽象层，可以接管各种数据访问方式的特定事务管理，然后以统一的方式管理和控制。

另外，如果使用Spring的IoC容器来构建应用，那么IoC容器将在以上所提到的各个方面的基础上，为应用程序提供更多附加值。

Spring框架提供集成支持的ORM解决方案包括Hibernate、iBATIS、JDO、Oracle Toplink、Apache OJB等。不过，随着Spring版本的升级，Spring核心包里对各种ORM解决方案的支持范围在逐渐缩小，对旧有版本ORM解决方案的支持逐步转给了Spring Modules子项目。到Spring 2.5之后，Spring核心包中提供的ORM集成支持只包括Hibernate 3、iBATIS 2、JDO、JPA以及Oracle的Toplink。我们不会详细介绍所有这些ORM的集成情况，因为在集成的方式上，几乎可以用"千篇一律"来形容。但是，Spring的集成方式比较统一并不意味着所有的ORM解决方案都是一样的。我们应该根据具体的场景，来决定使用何种ORM解决方案，需求上的因素也好，经济上的因素也罢，甚至技术上的因素，等等，而这些好像都与Spring对ORM解决方案的集成没有太多关系。Spring对ORM的集成，只是为了让我们能够在使用ORM的过程中活得更加舒服一些。

以下内容将主要围绕Spring对Hibernate 3和iBATIS 2这两种ORM方案的集成来进行讲述，最后会

① 第五部分将详细讲述统一的数据访问事务管理及控制方式，本章只关注数据访问相关内容，所以，有关统一的事务管理相关内容在本章不会涉及。

对其他ORM方案的集成做一定的概述，以便在你要使用这些ORM方案的时候，知道如何从Spring这里获得支持。现在，就让我们开始这趟Spring中对各种ORM解决方案的集成之旅吧！

15.1　Spring 对 Hibernate 的集成

在当今Java的各种ORM解决方案中，Hibernate凭借其先期的优势以及后期的持续跟进，无疑已经成为基于ORM进行数据访问的事实标准。我在2004年开始接触Hibernate，并应用于后继项目的开发。虽然当时只有2.1版本，但是我已经深刻感受到Hibernate的强大。随着Hibernate 3的发布，Hibernate加入了更多特性。

但是强大归强大，在具体项目实践中如何有效地使用Hibernate，却是一个值得探讨的问题。

注意　本节内容侧重于Spring框架如何对Hibernate的集成，所以，不会对Hibernate各种特性进行事无巨细的介绍，有关Hibernate的更多内容请参考官方文档，或者相关书籍——《Hibernate基础教程》（人民邮电出版社，2008.2）和《Hibernate 实战》（人民邮电出版社，2008.4），这两本书是优秀的Hibernate著作。

15.1.1　旧日“冬眠”①时光

现在说起三四年前的Hibernate项目实践，可能有人会嗤之以鼻。不过，“子曰：温故而知新”，从过去吸取养分然后继续前进才是我们的最终目的。即使在去年的项目中，我依然可以看到有的项目因为类似的Hibernate使用方式导致数据库泄漏事件，甚至在很长一段时间内才清理干净，这就更加坚定了我要旧事重提的决心。

需要先从Hibernate参考文档所提供的Session管理工具类说起，先看一下代码清单15-1的内容吧（摘自Hibernate 2.1.8参考文档）！

代码清单15-1　HibernateUtil定义代码摘录

```
public class HibernateUtil {
    private static Log log = LogFactory.getLog(HibernateUtil.class);
    private static final SessionFactory sessionFactory;
    static
    {
        try {
            // Create the SessionFactory
        sessionFactory =
            new Configuration().configure().buildSessionFactory();
        } catch (Throwable ex) {
            log.error("Initial SessionFactory creation failed.", ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static final ThreadLocal session = new ThreadLocal();

    public static Session currentSession() throws HibernateException {
        Session s = (Session) session.get();
        // Open a new Session, if this Thread has none yet
        if (s == null) {
            s = sessionFactory.openSession();
```

① 将冬眠加引号是要表示双关的意思，中学语文好像应该提到过引号的这种用途吧？总之，看官您心知肚明就行了！

```
            session.set(s);
        }
        return s;
    }

    public static void closeSession() throws HibernateException {
        Session s = (Session) session.get();
        session.set(null);
        if (s != null)
        s.close();
    }
}
```

HibernateUtil的作用是对Session的初始化，获取以及释放进行统一的管理。坦诚地说，只从HibernateUtil本身来说，我们没有太多可以指摘的，但是，当大部分人都按照代码清单15-2中示例代码的样子来使用HibernateUtil的时候，问题就比较容易出现了。

代码清单15-2 HibernateUtil一般使用代码示例

```
Session session = HibernateUtil.currentSession();

Transaction tx= session.beginTransaction();

Cat princess = new Cat();
princess.setName("Princess");
princess.setSex('F');
princess.setWeight(7.4f);
session.save(princess);

tx.commit();

HibernateUtil.closeSession();
```

仅关注这段使用代码好像没有什么不妥之处，可是，以同样的方式铺开到整个团队的开发中的话，就真的不妥了。

对于一个团队来说，工作的分配通常是按照功能模块划分的，而不是按照应用程序的分层划分的，这就意味着，每个开发人员通常要负责自己模块中每层的各种功能实现，具体点儿说就是，每个开发人员都需要负责自己功能模块中的DAO以及Service层对象开发。为了能够给每个开发人员开发DAO以及Service对象提供适度的“公共设施”，我们通常会将Hibernate的使用进行适当的封装，将公用的一些行为抽象到父类中定义，这样，我们就有了类似代码清单15-3所示的DAO父类实现（忽略了异常处理和其他细节）。

代码清单15-3 基于Hibernate的DAO父类定义代码示例

```
public abstract class AbstractHibernateDao {

    private Session session = null;
    private Transaction transaction = null;

    public AbstractHibernateDao() throws HibernateException
    {
        session = HibernateUtil.currentSession();
    }

    public Session getCurrentSession()
    {
```

```
        return this.session;
    }

    public Transaction beginTransaction()
    {
        return getCurrentSession().beginTransaction();
    }

    public void commit()
    {
        if(this.transaction != null)
            this.transaction.commit();
    }

    public void rollbck()
    {
        if(this.transaction != null)
            this.transaction.rollback();
    }

    public void closeSession()
    {
        HibernateUtil.closeSession();
    }
}
```

开发人员要实现DAO，需要继承该AbstractHibernateDao，并在构造方法中调用父类的构造方法。这样一旦子类实例化完毕，即可获得相应的Session进行数据访问，如代码清单15-4所示。

代码清单15-4 AbstractHibernateDao子类化定义代码示例

```
public class FooHibernateDao extends AbstractHibernateDao implements IBarDao {
    public FooHibernateDao()
    {
        super();
    }

    public void saveOperation(Object po)
    {
        getCurrentSession().save(po);
    }
    ....
}
```

因为事务控制通常在Service层，所以FooHibernateDao中没有过多涉及特定的事务API以及Session管理的代码。不过，在仔细看加入这些相关逻辑的Service实现类之后，就会发现以这种方式使用Hibernate会是一种多么糟糕的体验。代码清单15-5给出的就是这样一个Service实现类。

代码清单15-5 加入Hibernate的事务和Session管理相关代码之后的Service实现代码示例

```
public class FooService {


    private FooHibernateDao dao;

    public void someBusinessLogic() throws Exception
    {
        try
        {
```

```
            getDao().beginTransaction();

            Object mockPO = new Object();
            getDao().saveOperation(mockPO);

            getDao().commit();
        }
        catch(HibernateException e)
        {
            getDao().rollbck();
            throw e;
        }
        catch(Exception e)
        {
            getDao().rollbck();
            throw e;
        }
        finally
        {
            getDao().closeSession();
        }
    }

    ...

    public FooHibernateDao getDao() {
        return dao;
    }

    public void setDao(FooHibernateDao dao) {
        this.dao = dao;
    }
}
```

如果只是一个类或一个方法，即使这样处理，只要谨慎一些，还是可以保证程序的正常运行的。可是，在团队中充斥着各色开发人员的时候，当编程规范不能得到严格执行的时候，这样的Hibernate使用方式以及类似结构的程序设计，对于整个团队来说都不是一件容易的事情。要关注事务管理，同时也要关注资源（Session）的管理，还得适度进行异常的处理，也许每个开发人员都精明能干，但当所有这一切凑在一起的时候，就难免会有疏漏了。这其实也不难解释，以这样的实践方式来使用Hibernate为什么会时不时发生资源泄漏之类的事故了。

当然，以上示例只是抽象出来的，以分散的Hibernate API封装，来使用Hibernate进行数据访问的方式中的一种。实际上，只是同时关注数据访问中的多个方面，并且没有合适的封装方式来使用Hibernate。项目中类似于这样的代码应该并不少见，而这直接影响到软件产品的质量，这才是我们应该重点关注的。所以，我们应该寻求新的Hibernate实践，将开发人员从复杂的数据访问API的使用中解放出来，使其能够更加高效地工作，从而促进整个软件产品质量的提高。

15.1.2 “春天”里的“冬眠”

鉴于Hibernate API的使用，在具体项目实践过程中，也存在类似于JDBC API使用中资源管理以及异常的处理之类的普遍问题。Spring在`JdbcTemplate`的成功理念下，对Hibernate的使用以相同的方式进行了封装，从而使得我们不用在资源管理以及异常处理方面投入过多的精力。同时，Spring对所有

的数据访问技术相关的事务管理，也通过AOP的形式剥离出去，进一步避免了过多的方面纠缠在一起的困境。

1. HibernateTemplate的登场

对于旧日的Hibernate操作的封装方式来说，没有能够对Session资源的管理，以及数据访问异常进行更为集中合理的处理。所以，Spring提供了`HibernateTemplate`对Hibernate的使用进行模板化封装，并且在模板方法内部统一进行数据访问异常的处理。

- 基于`HibernateTemplate`的Session资源管理

Session是使用Hibernate与关系数据库进行数据访问的“纽带”，所有的数据访问操作必须经由Session的支持才能完成。对于日常开发来说，我们应该更多关注的是如何使用Session进行数据访问，而至于Session的获取以及释放等资源管理问题，则应该尽可能从频繁的数据访问代码中解脱出去，以避免过多方面的纠缠。过去的Hibernate使用方式，就是因为每个开发人员都需要同时关注Session资源的管理，以及具体的基于Session的数据访问逻辑，才会出现稍有不慎就出现资源泄漏的情况。

`HibernateTemplate`统一对Session的获取以及释放等管理逻辑进行封装，将Session管理尽量保持在一处进行。而对于不同的数据访问需求，`HibernateTemplate`提供了`HibernateCallback`回调接口，以便调用方可以根据各自的数据访问需求进行定制，完全不会限制Hibernate任何功能的发挥。

与`JdbcTemplate`的实现相似，`HibernateTemplate`所有的模板方法以一个主要的模板方法为中心，该方法通过`HibernateCallback`回调接口为调用方公开Session资源以进行数据访问。代码清单15-6是这个主要的模板方法定义。

代码清单15-6　`HibernateTemplate`核心模板方法定义摘录

```
public Object execute(HibernateCallback action, boolean exposeNativeSession) throws
  DataAccessException {
    Assert.notNull(action, "Callback object must not be null");

    Session session = getSession();
    boolean existingTransaction = (!isAlwaysUseNewSession() && ➥
    (!isAllowCreate() || SessionFactoryUtils.isSessionTransactional(session, ➥
    getSessionFactory())));
    if (existingTransaction) {
        logger.debug("Found thread-bound Session for HibernateTemplate");
    }

    FlushMode previousFlushMode = null;
    try {
        previousFlushMode = applyFlushMode(session, existingTransaction);
        enableFilters(session);
        Session sessionToExpose = (exposeNativeSession ? session : createSessionProxy(session));
        Object result = action.doInHibernate(sessionToExpose);
        flushIfNecessary(session, existingTransaction);
        return result;
    }
    catch (HibernateException ex) {
        throw convertHibernateAccessException(ex);
    }
    catch (SQLException ex) {
        throw convertJdbcAccessException(ex);
    }
    catch (RuntimeException ex) {
        // Callback code threw application exception...
        throw ex;
```

```
        }
        finally {
            if (existingTransaction) {
                logger.debug("Not closing pre-bound Hibernate Session after
                  HibernateTemplate");
                disableFilters(session);
                if (previousFlushMode != null) {
                    session.setFlushMode(previousFlushMode);
                }
            }
            else {
                // Never use deferred close for an explicitly new Session.
                if (isAlwaysUseNewSession()) {
                    SessionFactoryUtils.closeSession(session);
                }
                else {
        SessionFactoryUtils.closeSessionOrRegisterDeferredClose(session,
          getSessionFactory());
                }
            }
        }
    }
```

因为该模板方法要处理的方面很多，所以，看起来比较复杂。不过，让我们先把事务管理相关的代码搁置一边，对该模板方法进行简化，这样就比较容易看出该模板方法的真面目了。简化后的核心模板方法定义见代码清单15-7。

代码清单15-7　简化的HibernateTemplate核心模板方法定义代码示例

```
public Object execute(HibernateCallback action, boolean exposeNativeSession) throws ➥
DataAccessException {
    Assert.notNull(action, "Callback object must not be null");

    Session session = getSession();
    try {
        Object result = action.doInHibernate(session);
        return result;
    }
    catch (HibernateException ex) {
        throw convertHibernateAccessException(ex);
    }
    catch (SQLException ex) {
        throw convertJdbcAccessException(ex);
    }
    catch (RuntimeException ex) {
        // Callback code threw application exception...
        throw ex;
    }
    finally {
        closeSession(session);
    }
}
```

实际上，去除事务相关的代码，为Hibernate操作提供一个模板方法就这么简单，只要将Session资源的管理纳入一个模板方法，而不是让它任意的散落到代码各处，我们就可以在很大程度上避免Session资源泄漏的危险。

如果愿意，我们可以在这个主要模板方法的基础上，为HibernateTemplate添加更多的数据访问操作，所有之后添加的模板方法最终都通过该主要模板方法执行数据访问。当然，这部分工作Spring也已经为我们做了，基本上也不需要我们自己劳神。

● 特定于Hibernate的数据访问异常转译

在简化后的HibernateTemplate核心模板方法中，除了对Session资源管理的代码，还需要我们关注的就是数据访问异常的处理。

Hibernate 3之前的Hibernate数据访问异常类型是checked exception，自然而然地就会出现本章开头的一幕，客户端需要根据特定的数据访问方式，捕获并处理特定于数据访问技术的异常类型。Spring从开始对Hibernate 2提供支持的时候，就提供了SessionFactoryUtils工具类，帮助实现从HibernateException到Spring统一异常体系的转译。

随着越来越多的开发人员对unchecked exception在数据访问领域存在合理性的认同，Hibernate 3之后的HibernateException也改为了unchecked exception类型，但将unchecked exception的HibernateException转译到Spring的异常体系依然是值得做的一件事情。

SessionFactoryUtils类提供了convertHibernateAccessException静态方法进行HibernateException到Spring异常体系的转译（如下）：

```
public static DataAccessException convertHibernateAccessException(HibernateException ex)
```

HibernateTemplate内部在实现HibernateException的异常转译的时候，最终也是委派该方法进行的。至于模板方法中需要处理的SQLException，HibernateTemplate则将这部分的转译工作转交给了SQLExceptionTranslator。通过学习前面的JdbcTemplate，我们对其并不陌生。

如果使用HibernateTemplate进行基于Hibernate的数据访问，我们无需关心具体的异常转译，因为HibernateTemplate内部可以很好地处理这一点。不过，即使不能够使用HibernateTemplate，也没关系，在基于Hibernate数据访问代码中，依然可以借助SessionFactoryUtils的convertHibernateAccessException方法进行异常的转译。

● 基本数据访问操作

HibernateOperations接口定义了能够通过HibernateTemplate使用的所有基于Hibernate的数据访问方法，HibernateTemplate实现了该接口。HibernateOperations的定义实在是太长，这里不详细罗列，在使用的过程中可以参考HibernateTemplate或者HibernateOperations的Javadoc文档。

要使用HibernateTemplate进行数据访问，只要为其提供一个可以使用的SessionFactory实例即可。通过编程的方式提供还是通过Spring的IoC容器进行注入，完全可以根据具体应用场景来决定。

(1) 基于HibernateTemplate的查询。HibernateTemplate为查询操作提供的模板方法比较丰富，主要分为如下4组。

- **get模板方法**。get组的模板方法，可以让我们通过指定的主键从数据库中获取相应的对象数据。如果指定的主键没有对应的数据，该组方法将返回null，这与Session的get方法的行为是一致的。
- **load模板方法**。与get组模板方法相近，load模板方法也是通过指定的主键到数据库加载相应的对象数据。但如果指定的主键没有对应的数据，该方法将抛出数据访问异常，而不是返回null，与Session中的load方法的行为也是一致的。
- **find模板方法**。find组的模板方法还可以细分为如下几种。
  - **find模板方法**。可以通过find组的模板方法传入相应HQL语句进行查询，该组模板方法将

根据指定的HQL语句执行查询，将查询结果以List的形式返回。使用find组模板方法指定的HQL中如果有占位符的话，只能通过"?"指定。

- **findByNamedParam模板方法**。与find模板方法的行为相近，只不过，可以指定使用命名参数占位符的HQL语句进行查询。
- **findByNamedQuery模板方法**。NamedQuery是指定义于Hibernate的映射文件中的HQL查询定义，findByNamedQuery模板方法对应的HQL中的参数占位符使用"?"。
- **findByNamedQueryAndNamedParam模板方法**。是findByNamedQuery模板方法的补充，可以使用命名参数占位符替代"?"占位符来指定HQL查询语句。

- **iterate模板方法**。iterate模板方法将按照方法指定的HQL查询语句以及相关参数执行查询，查询结果以迭代器（iterator）的形式返回，而不是List。在一次查询数据量比较大且占用系统过多资源的情况下，通常使用iterate模板方法进行查询，直接使用Session的iterate方法进行查询也应出于同样目的。

如果要对fx_news中指定的记录进行查询，可以使用如下代码进行：

```
SessionFactory sessionFactory = ...;
HibernateTemplate hibernateTemplate = new HibernateTemplate(sessionFactory);
Object po = hibernateTemplate.get(FXNewsBean.class,new Integer(1));
if(po == null)
    // 没有查找到指定记录
else
    // 处理FXNewsBean相应信息
```

要让这段代码运行，前提是Hibernate的映射文件以及PO（Persistent Object）等准备就绪。有关如何配置Hibernate进行数据访问的内容，请参照Hibernate的参考文档以及相关书籍。

至于其他几组查询方法，就留待你自己去验证和使用吧！

(2) 基于HibernateTemplate的更新。我们这里的更新属于广义上的更新，包括数据的插入、更新和删除操作。HibernateTemplate为这些具体的操作同样提供了完善的模板方法支持，如下所述。

- **save组模板方法**。包括save和saveOrUpdate两组重载的模板方法，前者将在本地构建的持久化对象插入数据库，后者则根据情况来决定是插入还是更新方法参数所提供的持久化对象。save和saveOrUpdate的更多细节，请参阅相关文档。
- **update组模板方法**。对应当前Session的持久化对象的部分信息被修改后，可以通过update方法将修改后的信息更新到后台数据库，以保持持久化对象与数据库数据的一致性。
- **delete组模板方法**。将指定的持久化对象所对应的信息记录，清除出后台存储媒介，通常是从数据库中删除，deleteAll()方法允许一次指定多个将要被删除的持久化对象实例。

有了这些模板方法的支持，基于Hibernate的数据更新就更加令人愉悦了。不信？见代码清单15-8。

代码清单15-8 基于HibernateTemplate的数据更新操作代码示例

```
DataFieldMaxValueIncrementer incrementer = ...;
String pk = incrementer.nextStringValue();

FXNewsBean po = new FXNewsBean();
po.setNewsId(pk);
po.setNewsTitle("...");
po.setNewsBody("...");

SessionFactory sessionFactory = ...;
HibernateTemplate hibernateTemplate = new HibernateTemplate(sessionFactory);
```

15

```
hibernateTemplate.save(po);

po.setNewsBody("new content");
hibernateTemplate.update(po);

hibernateTemplate.delete(po);
```

2. Spring中的SessionFactory的配置及获取

Hibernate的SessionFactory，就好像JDBC的DataSource，它是所有数据访问资源的发源地。只有获取了SessionFactory的支持，后继的数据访问才能继续，这也是为什么HibernateTemplate必须拥有一个SessionFactory才能工作的原因。

要配置并构建一个SessionFactory，最简单的方法就是直接通过编程的方式获得，如下所示：

```
Configuration cfg = new Configuration()
    .addResource("mappingResource.hbm.xml")
    ...
    .setProperty("hibernate.dialect", "org.hibernate.dialect.MySQLInnoDBDialect")
    ...;
SessionFactory sessionFactory = cfg.buildSessionFactory();
// sessionFactory 处于可用状态
```

不过，这里要说的是在Spring中如何配置和获取SessionFactory，这些基本的方式就先放在一边不谈。

- LocalSessionFactoryBean

Spring采用FactoryBean的形式对SessionFactory的配置和获取进行封装，org.springframework.orm.hibernate3.LocalSessionFactoryBean[①]将是我们在Spring中配置和获取SessionFactory最为常用的方式。LocalSessionFactoryBean作为一个FactoryBean实现，其对应的getObject()方法将返回SessionFactory类型的对象。我们可以通过LocalSessionFactoryBean配置Hibernate数据访问相关的所有资源，包括DataSource、配置文件位置、映射文件位置甚至于处理LOB数据的LobHandler。

代码清单15-9给出了LocalSessionFactoryBean在Spring的IoC容器中的常见配置情况。

代码清单15-9　IoC容器中使用LocalSessionFactoryBean的配置代码示例

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" ➥
destroy-method="close">
    <property name="url">
        <value>${jdbc.url}</value>
    </property>
    <property name="driverClassName">
        <value>${jdbc.driver}</value>
    </property>
    <property name="username">
        <value>${jdbc.username}</value>
    </property>
    <property name="password">
        <value>${jdbc.password}</value>
    </property>
</bean>

<bean id="sessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
```

① Hibernate 2的版本对应org.springframework.orm.hibernate2.LocalSessionFactoryBean。

```
    <property name="dataSource" ref="dataSource"/>
    <property name="configLocation">
        <value>org.spring21..hibernate.cfg.xml</value>
    </property>
</bean>
```

Spring为了能够以统一的方式对各种数据访问方式的事务管理进行抽象，在通过LocalSessionFactoryBean构建SessionFactory的时候，使用的是容器内定义的DataSource定义，而不是使用Hibernate内部的ConnectionProvider。实际上，在构建SessionFactory的实际过程中，Spring将根据传入的DataSource来决定为将要构建的SessionFactory提供什么样的ConenctionProvider实现。这些ConnectionProvider实现包括LocalDataSourceConnectionProvider以及TransactionAwareDataSourceConnectionProvider，我们可以在LocalSessionFactoryBean同一包下面找到它们。

在代码清单15-9所示的容器配置文件中，只通过configLocation就完成了整个的LocalSessionFactoryBean的配置，这是因为通常的hibernate.cfg.xml配置文件中已经包含了几乎所有必需的配置信息，包括映射文件资源、各种Hibernate配置参数等。不过，也可以像代码清单15-10所示，通过LocalSessionFactoryBean单独指定这些配置项。

代码清单15-10 单独指定各种配置项的LocalSessionFactoryBean配置示例

```
<bean id="sessionFactory" class="org.springframework.orm.hibernate3.
  LocalSessionFactoryBean">
    <property name="dataSource" ref="dataSource"/>
    <property name="mappingResources">
        <list>
            <value>cn/spring21/../mapping1.hbm.xml</value>
            <value>cn/spring21/../mapping2.hbm.xml</value>
            <value>cn/spring21/../mapping3.hbm.xml</value>
            <value>...</value>
        </list>
    </property>
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
            <prop key="...">...</prop>
        </props>
    </property>
</bean>
```

当然，如果没有特殊目的，将所有的Hibernate配置信息保存到Hibernate指定的配置文件中统一管理，才是比较合适的做法。

更多LocalSessionFactoryBean的可配置项请参考Javadoc文档。

- AnnotationSessionFactoryBean

Hibernate从2.*x*版本就支持基于XML的映射文件来定义映射信息。不过，ORM的映射信息可以通过多种方式来表达。随着Java 5的普及，越来越多的开发人员喜欢使用Java 5中的注解来记录一些元数据信息。Hibernate在这一点上自然也不落后，通过Hibernate注解的支持，Hibernate中使用的映射信息也可以通过注解的形式来表达。而AnnotationSessionFactoryBean就是为那些通过注解获取映射信息的SessionFactory的配置以及构建而准备的。

AnnotationSessionFactoryBean是在LocalSessionFactoryBean基础上构建的，除了可以指定LocalSessionFactoryBean的配置项之外，AnnotationSessionFactoryBean追加定义了几个专门

15

用于获取注解元数据定义类的配置项，包括annotatedClasses、annotatedPackages以及configurationClass。代码清单15-11是AnnotationSessionFactoryBean的简单配置实例。

代码清单15-11　AnnotationSessionFactoryBean配置代码示例

```
<bean id="sessionFactory" class="org.springframework.orm.hibernate3.annotation.
  AnnotationSessionFactoryBean">
    <property name="dataSource" ref="dataSource">
      <property name="annotatedClasses">
        <list>
             <value>cn.spring21..mapping.AnnotationMappingClass1</value>
              <value>cn.spring21..mapping.AnnotationMappingClass1</value>
        </list>
      </property>
      <property name="annotatedPackages">
        <list>
              <value>cn.spring21..package</value>
        </list>
      </property>
</bean>
```

注意　有关可用的注解类型可以参照Hibernate Annotation项目文档，因为这属于Hibernate部分的内容，此处不会过多讲述。

因为标注有映射信息注解的类也可以在hibernate.cfg.xml之类的统一的配置文件中配置，所以，这种情况下，也可以不使用annotatedClasses之类的属性，直接使用一个configLocation属性配置即可搞定。

- 通过JNDI获取SessionFactory

除了可以在本地直接构建SessionFactory实例使用，或者通过Spring的IoC容器注册LocalSessionFactoryBean或者AnnotationSessionFactoryBean来获取相应的SessionFactory支持，我们也可以将SessionFactory绑定到服务器端的JNDI服务上，当需要使用的时候，可以通过JNDI查询来获得。

但将SessionFactory通过JNDI或者JCA绑定到具体的容器中是否必要却有值得商榷的地方。通过LocalSessionFactoryBean或者AnnotationSessionFactoryBean，在Spring容器中本地定义并获取SessionFactory，对于通常的应用程序来说是最为合适的方式，而通过JNDI绑定到具体容器则无法获取任何益处。 SessionFactory所持有的配置信息以及资源很大一部分是应用程序独有的，比如Domain Object，这不同于DataSource，将这些独有的信息绑定到JNDI再去获取，就好像你自己家的电视机必须放到一个地方去保管，当你要看的时候，还需要到那个地方去申请领回一样。

但对于某些场景来说，通过JCA来注册SessionFactory确实可以获得一定的优势，如结合EJB使用的时候。

总之，是否要通过JNDI来获取SessionFactory应该根据具体应用程序的场景来决定，但一般情况下，基于Spring IoC容器的SessionFactory配置及构建，就已经是最佳方案了。

3. HibernateDaoSupport

与JdbcDaoSupport所完成的使命相同，即使我们可以在DAO中直接使用HibernateTemplate进行数据访问，但也没有必要每个开发人员在各自的每个DAO实现类中都声明一个HibernateTemplate的实例。通常我们会提供一个DAO的基类，其中声明HibernateTemplate的支持，这样，子类只需要使用父类提供的HibernateTemplate进行数据访问操作即可。

HibernateDaoSupport的出现让我们免于重新去发明这种基于HibernateTemplate的DAO基类的轮子。在实现基于HibernateTemplate的DAO实现类的时候，我们直接继承HibernateDaoSupport就可以获得HibernateTemplate的数据访问支持，如代码清单15-12所示。

代码清单15-12 HibernateDaoSupport应用代码示例

```
public class FooHibernateDao extends HibernateDaoSupport implements IFooDao
{
    public void insertNews(FXNewsBean newsBean)
    {
        getHibernateTemplate().save(newsBean);
    }

    public void deleteNews(Object po)
    {
        getHibernateTemplate().delete(po);
    }
    ...
}
```

HibernateDaoSupport还为子类公开了异常转译的方法，如下所示：

```
protected final DataAccessException convertHibernateAccessException(HibernateException ex)
```

即使在具体的DAO实现类中使用原始的Hibernate API进行数据访问，只要我们继承了HibernateDaoSupport，也可以获得HibernateException到Spring统一异常体系的转译支持。

15.2 Spring对iBATIS的集成

iBATIS（http://ibatis.apache.org/）是我近一年多使用最多的ORM解决方案。如果将Hibernate比做自动步枪，而将JDBC比做手动步枪的话，那么iBATIS就得算是半自动步枪了。iBATIS并没有像Hibernate之类的ORM解决方案那样提供完备的ORM特性，包括对象查询语言、透明持久化等，但iBATIS却以其他的优势在这百花争艳的ORM武林中占有一席之地。相对于其他完备的ORM产品来说，iBATIS学习曲线很低，你以及你的团队只要精通SQL，那基本上就没有太多问题。上手快绝不是吹出来的，因为我亲身经历了带领一个之前没有任何iBATIS经验的团队的开发过程。如果想以更加灵活的方式使用JDBC，而且不想引入过于复杂的ORM产品却想使用一定的ORM特性，那么，iBATIS应该是我们的最佳选择。

不过，既然iBATIS也是一种依赖于数据资源的访问技术，那么，通常也避免不了像JDBC以及Hibernate在具体的使用过程中所遇到的资源管理、异常处理等各种方面的问题。所以，为了能够在实际的开发过程中使用最佳的iBATIS实践方式，先让我们一起来探索一下iBATIS最佳实践的“前生”和“今世”吧！

15.2.1 iBATIS实践之“前生”篇

在iBATIS中，通常是通过com.ibatis.sqlmap.client.SqlMapClient进行数据访问的。当然，在使用之前，我们需要先构建一个可用的实例，如代码清单15-13所示。

代码清单15-13 一般情况下SqlMapClient的初始化代码示例

```
    Reader reader = null;
SqlMapClient sqlMap = null;
try
```

```
{
    String resource = "com/ibatis/example/sqlMap-config.xml";
    reader = Resources.getResourceAsReader (resource);
    sqlMap = SqlMapClientBuilder.buildSqlMapClient(reader);
}
catch (IOException e)
{
    e.printStackTrace(); // 不要这样做
}
// sqlMap现在处于可用状态
```

与其他ORM一样，iBATIS同样需要一个总的配置文件来获取具体的Sql映射文件。根据该配置文件所提供的配置信息，就可以使用SqlMapClientBuilder来构建一个可用的SqlMapClient实例了。而这之后，是通过Singleton方式还是工厂方式将这个实例公开给系统，就完全由你来决定了。

通常，我们可以通过以下三种方式来使用SqlMapClient进行数据访问。

基于SqlMapClient的自动提交事务型简单数据访问。应该说，SqlMapClient对于资源的管理已经进行了足够的封装，使用SqlMapClient进行自动提交事务性质的数据访问，根本就不用考虑太多资源管理的问题，例如：

```
Map parameters = new HashMap();
parameters.put("parameterName",value);
...
Object result = sqlMap.queryForObject("System.getSystemAttribute", parameters);
```

基于SqlMapClient的非自动提交事务型数据访问。虽然直接使用SqlMapClient进行数据访问可以暂时不考虑资源管理问题，但是在数据访问代码中加入事务控制代码以及异常处理代码之后，事情看起来就不像开始那么清爽了（见代码清单15-14）。

代码清单15-14 引入事务管理和异常处理的SqlMapClient使用代码示例

```
try
{
    sqlMap.startTransaction();
    sqlMap.update("mappingStatement");
    sqlMap.commitTransaction();
} catch (SQLException e) {
    e.printStackTrace(); // 不要这样做
}
finally
{
    try {
        sqlMap.endTransaction();
    } catch (SQLException e) {
        e.printStackTrace(); // 不要这样做
    }
}
```

基于SqlMapSession的数据访问。另外，也可以从SqlMapClient中获取SqlMapSession，由自己来管理数据访问资源以及相关的事务控制和异常处理，如代码清单15-15所示。

代码清单15-15 使用SqlMapSession进行数据访问的代码示例

```
SqlMapSession session = null;
try
{
    session = sqlMap.openSession();
```

```
        session.startTransaction();
        session.update("");
        session.commitTransaction();

    } catch (SQLException e) {
        e.printStackTrace(); // 不要这样做
    }
    finally
    {
        if(session != null)
        {
            try {
                session.endTransaction();
            } catch (SQLException e) {
                e.printStackTrace(); // 不要这样做
            }
            session.close();
        }
    }
```

这种方式可能获取少许的性能优势，不过通常与第二种直接使用SqlMapClient进行数据访问没有太大分别，只不过把一些原来由SqlMapClient管理的任务拿来自己管理而已。

实际上，可以看出，除了基于SqlMapClient的自动提交事务性的数据访问方式之外，其他两种方式在引入事务管理和异常处理甚至资源管理等方面的代码之后，整个的数据访问代码就变得有些难以管理了。如果让每个开发人员在实现DAO的时候，或者在Service层管理事务的时候都使用与上面几乎相同的代码，而不是采用某种方式统一管理一下，那么整个系统的稳定性和产品质量就可能存在问题了。

15.2.2 iBATIS 实践之“今世”篇

因为Spring在集成iBATIS的时候，还要考虑将它的事务控制也纳入Spring统一的事务管理抽象层，所以使用了基于SqlMapSession的数据访问方式对iBATIS进行集成。这种方式更为灵活，可以将iBATIS内部可以直接指定的数据源以及事务管理器等“设备”转由外部提供，如使用Spring的IoC容器为其注入。

org.springframework.orm.ibatis.SqlMapClientTemplate是Spring为基于iBATIS的数据访问操作提供的模板方法类，我们可以通过SqlMapClientTemplate结合org.springframework.orm.ibatis.SqlMapClientCallback回调接口完成所有基于iBATIS的数据访问操作。SqlMapClientTemplate管理事务、异常处理、资源管理等方面的事情，而SqlMapClientCallback则使得开发人员专注于具体的数据访问逻辑。

1. SqlMapClientTemplate的实现

SqlMapClientTemplate中的execute(SqlMapClientCallback)方法是整个SqlMapClientTemplate实现的核心，如下定义：

```
public Object execute(SqlMapClientCallback action) throws DataAccessException
```

所有其他的模板方法都是在该模板方法的基础上为了进一步提供便利而提供的。

execute(SqlMapClientCallback)模板方法以统一的方式，对基于iBATIS的数据访问操作进行了封装并于一处管理，可谓集资源管理、异常处理以及事务控制机能于一身，其定义的代码摘录如代码清单15-16所示。

15

代码清单15-16 SqlMapClientTemplate核心模板方法定义代码摘录

```
public Object execute(SqlMapClientCallback action) throws DataAccessException {
    Assert.notNull(action, "Callback object must not be null");
    Assert.notNull(this.sqlMapClient, "No SqlMapClient specified");

    // We always needs to use a SqlMapSession, as we need to pass a Spring-managed
    // Connection (potentially transactional) in. This shouldn't be necessary if
    // we run against a TransactionAwareDataSourceProxy underneath, but unfortunately
    // we still need it to make iBATIS batch execution work properly: If iBATIS
    // doesn't recognize an existing transaction, it automatically executes the
    // batch for every single statement...

    SqlMapSession session = this.sqlMapClient.openSession();
    if (logger.isDebugEnabled()) {
        logger.debug("Opened SqlMapSession [" + session + "] for iBATIS operation");
    }
    Connection ibatisCon = null;

    try {
        Connection springCon = null;
        DataSource dataSource = getDataSource();
        boolean transactionAware = (dataSource instanceof TransactionAwareDataSourceProxy);

        // Obtain JDBC Connection to operate on...
        try {
            ibatisCon = session.getCurrentConnection();
            if (ibatisCon == null) {
                springCon = (transactionAware ?
                        dataSource.getConnection() : DataSourceUtils.doGetConnection
                          (dataSource));
                session.setUserConnection(springCon);
                if (logger.isDebugEnabled()) {
                    logger.debug("Obtained JDBC Connection [" + springCon + "] for iBATIS
                      operation");
                }
            }
            else {
                if (logger.isDebugEnabled()) {
                    logger.debug("Reusing JDBC Connection [" + ibatisCon + "] for iBATIS
                      operation");
                }
            }
        }
        catch (SQLException ex) {
            throw new CannotGetJdbcConnectionException("Could not get JDBC Connection", ex);
        }

        // Execute given callback...
        try {
            return action.doInSqlMapClient(session);
        }
        catch (SQLException ex) {
            throw getExceptionTranslator().translate("SqlMapClient operation", null, ex);
        }
        finally {
            try {
                if (springCon != null) {
                    if (transactionAware) {
                        springCon.close();
```

```
                }
                else {
                    DataSourceUtils.doReleaseConnection(springCon, dataSource);
                }
            }
        }
        catch (Throwable ex) {
            logger.debug("Could not close JDBC Connection", ex);
        }
    }

    // Processing finished - potentially session still to be closed.
}
finally {
    // Only close SqlMapSession if we know we've actually opened it
    // at the present level.
    if (ibatisCon == null) {
        session.close();
    }
}
}
```

该方法定义初看起来或许感觉繁杂，是因为Spring在集成iBATIS的时候，要考虑在整个框架内以统一的方式处理事务管理，才会出现这种初看起来不慎清晰的代码实现。

execute方法一开始通过给定的sqlMapClient获取相应的SqlMapSession:

```
SqlMapSession session = this.sqlMapClient.openSession();
```

我们说过Spring在集成iBATIS的时候使用的是基于SqlMapSession的数据访问方式，所以，这行代码很好理解。

从Connection ibatisCon = null;这行代码一直到// Execute given callback...这行注释之前，都可以看作是Spring在处理对事务管理的集成。Spring会根据当前的事务设置决定通过何种方式从指定的dataSource中获取相应的Connection供SqlMapSession使用。

之后的代码就比较容易理解了，模板方法会调用SqlMapClientCallback的回调方法来进行数据访问，如果期间出现SQLException，则通过提供的SQLExceptionTranslator进行异常转译。最后，合适地关闭所使用的数据库连接和SqlMapSession。

这实际上就是整个SqlMapClientTemplate的奥秘所在。

为了避免开发人员在一些通用的数据访问操作上提供几乎相同的SqlMapClientCallback实现，SqlMapClientTemplate同时在execute(SqlMapClientCallback)核心模板方法的基础上，提供了其他一些便利的数据访问模板方法。而这些模板方法在实现数据访问逻辑的时候，最终都会回到核心模板方法中，例如：

```
public Object queryForObject(final String statementName, final Object parameterObject)
        throws DataAccessException {

    return execute(new SqlMapClientCallback() {
        public Object doInSqlMapClient(SqlMapExecutor executor) throws SQLException {
            return executor.queryForObject(statementName, parameterObject);
        }
    });
}
```

到现在，你是不是应该更加了解SqlMapClientTemplate了呢？

15

2. SqlMapClientTemplate的使用

SqlMapClientTemplate底层依赖于com.ibatis.sqlmap.client.SqlMapClient来提供基于iBATIS的数据访问支持，所以，要使用SqlMapClientTemplate，最少需要提供一个SqlMapClient。

- **SqlMapClientTemplate的构建**

如果是通过编程的方式使用SqlMapClientTemplate，那么我们就可以像当初那样，直接使用iBATIS的方式，通过SqlMapClientBuilder构建相应的SqlMapClient实例，然后传给SqlMapClientTemplate，如下所示：

```
String     resource = "com/ibatis/example/sqlMap-config.xml";
Reader       reader = Resources.getResourceAsReader (resource);
SqlMapClient sqlMap = SqlMapClientBuilder.buildSqlMapClient(reader);
SqlMapClientTemplate sqlMapClientTemplate = new SqlMapClientTemplate(sqlMap);
// sqlMapClientTemplate准备就绪，可以使用
```

当然，只提供SqlMapClient实例的话，SqlMapClientTemplate将使用iBATIS配置文件内部指定的DataSource。我们也可以使用外部定义的DataSource，例如：

```
// 1. 定义DataSource
BasicDataSource dataSource = new BasicDataSource();
...
// 2. 定义SqlMapClient
...
SqlMapClient sqlMap = SqlMapClientBuilder.buildSqlMapClient(reader);
// 3. 创建SqlMapClientTemplate
SqlMapClientTemplate sqlMapClientTemplate = new SqlMapClientTemplate(dataSource,sqlMap);
// sqlMapClientTemplate准备就绪，可以使用
```

实际上，除了直接通过编程的方式配置和使用SqlMapClientTemplate，更多时候我们是通过IoC容器来配置和使用SqlMapClientTemplate的。在Spring的IoC容器中，我们使用org.springframework.orm.ibatis.SqlMapClientFactoryBean配置和获取相应的SqlMapClient，并注入给SqlMapClientTemplate使用。SqlMapClientFactoryBean也是一个FactoryBean实现，其getObject()方法返回的就是我们需要的SqlMapClient。

通过SqlMapClientFactoryBean配置和使用SqlMapClient以及SqlMapClientTemplate的情形如代码清单15-17所示。

代码清单15-17　SqlMapClientTemplate配置代码示例

```
<bean id="mainDataSource" class="org.apache.commons.dbcp.BasicDataSource" ➥
destroy-method="close">
    <property name="url">
        <value>${db.main.url}</value>
    </property>
    <property name="driverClassName">
        <value>${db.main.driver}</value>
    </property>
    <property name="username">
        <value>${db.main.user}</value>
    </property>
    <property name="password">
        <value>${db.main.password}</value>
    </property>
    ...
</bean>
```

```
<bean id="mainSqlMapClientTemplate" class="org.springframework.orm.ibatis.
  SqlMapClientTemplate">
    <property name="sqlMapClient">
        <bean class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
            <property name="dataSource">
                <ref bean="mainDataSource"/>
            </property>
            <property name="configLocation">
                <value>file:conf/ibatis/ibatis-config.xml</value>
            </property>
        </bean>
    </property>
</bean>
```

如果容器中的SqlMapClientFactoryBean仅限于一个SqlMapClientTemplate使用的话，可以像代码清单15-17所示的配置那样，将SqlMapClientFactoryBean配置为内部bean的形式。当然，这不是必须的。

- 使用SqlMapClientCallback进行数据访问

SqlMapClientCallback在整个Spring对iBATIS的集成中所处的地位仅次于SqlMapClientTemplate。通过SqlMapClientTemplate的execute(SqlMapClientCallback)模板方法结合SqlMapClientCallback，我们可以完成任何基于iBATIS的数据访问操作。

SqlMapClientCallback的定义很简单（通常Callback接口都很简单），通过doInSqlMapClient()方法公开了com.ibatis.sqlmap.client.SqlMapExecutor用于具体的数据访问操作，如下所示：

```
Object doInSqlMapClient(com.ibatis.sqlmap.client.SqlMapExecutor executor) throws
  SQLException
```

有了SqlMapExecutor，我们几乎就可以在iBATIS的世界里面想干什么就干什么，包括基本的查询、插入、更新、删除等基本数据访问操作，甚至是批量更新操作。

如果要向某一个数据库表批量更新数据，我们可以借助于SqlMapClientCallback所公开的SqlMapExecutor轻而易举地完成，如代码清单15-18所示。

代码清单15-18　使用SqlMapClientCallback进行数据的批量更新操作代码示例

```
protected void batchInsert(final List beans)
{
    sqlMapClientTemplate.execute(new SqlMapClientCallback() {
            public Object doInSqlMapClient(SqlMapExecutor executor) throws SQLException {
                executor.startBatch();
                Iterator iter = beans.iterator();
                while (iter.hasNext()) {
                        YourBean bean = (YourBean)iter.next();
                        executor.insert("Namespace.insertStatement", bean);
                }
                executor.executeBatch();
                return null;
            }
      });
}
```

15

至于其他基于iBATIS的基本的数据访问操作，也可以用类似的方式为SqlMapClientTemplate提供相应的SqlMapClientCallback实现。不过，通常情况下，我们不需要直接这么做，因为

SqlMapClientTemplate为我们考虑得更多。

- 基于SqlMapClientTemplate的基本数据访问操作

如果要使用SqlMapClientTemplate进行基于iBATIS的一些基本的数据访问操作，那么不需要每次都提供一个SqlMapClientCallback实现，然后通过execute(SqlMapClientCallback)来执行这些操作。为了便利起见，SqlMapClientTemplate为这些基本数据操作提供了足以满足需要的多组模板方法，我们所要做的，只是根据需要来选用这些便利的模板方法而已。

所有的SqlMapClientTemplate中定义的数据访问操作方法都在org.springframework.orm.ibatis.SqlMapClientOperations接口中定义，也包括这些便利的模板方法。实际上，在开发过程中，借助于IDE的支持，只要稍微参照一下Javadoc就能娴熟地使用这些模板方法。代码清单15-19给出了其中部分模板方法的使用代码示例。

代码清单15-19　使用SqlMapClientTemplate进行数据访问代码示例

```
SqlMapClientTemplate sqlMapClientTemplate = ...;
Object parameter = ...;
// 1. 插入数据
sqlMapClientTemplate.insert("insertStatementName",parameter);
// 2. 更新
int rowAffected = sqlMapClientTemplate.update("updateStatementName");
// 3. 删除
int rowAffected = sqlMapClientTemplate.delete("deleteStatementName",parameter);
// 4. 查询
Object result = sqlMapClientTemplate.queryForObject("selectStatementName");
List resultList = sqlMapClientTemplate.queryForList("selectStatementName");
Map resultMap = sqlMapClientTemplate.queryForMap("selectStatementName");
sqlMapClientTemplate.queryWithRowHandler("hugeSelect", new RowHandler(){
    public void handleRow(Object valueObject) {
        ResultBean bean = (ResultBean)valueObject;
        process(bean);
}});
```

更多信息可以参照SqlMapClientTemplate的Javadoc和iBATIS的参考文档以及相关书籍。

3. SqlMapClientDaoSupport

与集成其他数据访问方式同等待遇，如果要使用iBATIS实现相应的DAO的话，Spring为我们提供了org.springframework.orm.ibatis.support.SqlMapClientDaoSupport作为整个DAO层次体系的基类。所有开发人员在使用iBATIS进行DAO开发的时候，直接继承SqlMapClientDaoSupport类即可获得SqlMapClientTemplate的数据访问支持，如代码清单15-20所示。

代码清单15-20　SqlMapClientDaoSupport应用代码示例

```
public class FooSqlMapClientDao extends SqlMapClientDaoSupport implements IFooDao
{
    public void update(YourBean bean)
    {
        getSqlMapClientTemplate.update("updateStatementName",bean);
    }

    public Object get(String pk)
    {
        return getSqlMapClientTemplate().queryForObject("selectStmtName",pk);
    }
    ...
}
```

当然，如果愿意，直接为相应的DAO注入SqlMapClientTemplate来使用也没有人会反对，尤其是在系统中旧有的DAO基类已经存在的情况下。

15.3 Spring 中对其他 ORM 方案的集成概述

不管是从对各种ORM产品的集成理论以及集成方式上看，还是从对各种ORM产品集成关注点来看，Spring对各种ORM产品的集成几乎是一脉相承的。所以，对于Spring中其他几种ORM产品的集成情况，没有必要再重复几乎一样的理论和方式，因此以下内容仅作提点，不着更多笔墨。

回顾Spring对各种数据访问技术的集成，我们可以归纳出如下几点。

1. 集成理论和方式上

各种数据访问方式的管理和使用通过相应的模板方法类来统一建模，而具体的数据访问逻辑则统一由相应的回调接口提供，从而将数据访问资源的管理和具体的数据访问逻辑相分离。

2. 集成关注点上

Spring对各种数据访问技术的集成主要集中在以下三个关注点上。

- 数据访问资源管理，主要涉及两种管理对象，如下所述。
  - **连接工厂（ConnectionFactory）**。连接工厂代表的是创建数据访问会话资源的统一概念，通常可以通过特定的数据访问技术的支持来直接创建它们，也可以通过JNDI等服务获取已经创建并配置好的实例。对于JDBC来说，对应连接工厂概念的实体是DataSource，Hibernate是SessionFactory，iBATIS是SqlMapClient……
  - **连接（Connection）或者说会话资源**。连接是客户端与数据媒介进行数据通信的纽带，每次进行数据访问的时候，都需要从指定的连接工厂获得某个连接以完成本次数据访问操作，操作完成后关闭当前连接资源。对于JDBC来说，连接概念对应的是java.sql.Connection，对于Hibernate来说，对应的是Session，对于iBATIS是SqlMapSession......

  通常，Spring通过FactoryBean对相应的连接工厂资源的配置和创建进行封装，使得它们能够很好地集成到IoC容器中。对于连接资源的管理，则是由相应的模板方法类来处理。
- 特定数据访问异常的转译，将这些特定的数据访问异常转译为Spring统一的数据访问异常体系，从而使得客户端可以使用统一的方式透明的处理数据访问异常。
- 将特定于数据访问技术的事务管理，统一纳入到Spring的事务管理抽象层，使得我们可以用统一的方式来管理事务。最主要的是，通过Spring的事务管理抽象层，我们可以得到EJB 2时代只有通过相应的EJB Container才能获得的声明式事务支持。事务管理是一个大的话题，此处只介绍最初的两个关注点。

在此基础上，让我们来快速浏览一遍Spring支持的其他数据访问技术。

15.3.1 Spring 对 JDO 的集成

JDO（Java Data Object，网址为http://java.sun.com/jdo/）是Sun提出来的数据持久化规范。该规范从1.0版本发展到现在的2.0版本，虽然整个发展历程不如Hibernae那样突飞猛进，但也涌现了不少优秀的JDO产品实现，比如KodoJDO、JPOX（http://www.jpox.org/）等。现在JDO规范的发展主要由Apache JDO（http://db.apache.org/jdo/index.html）开源项目来推动。

1. Spring中的JDO资源管理

JDO中对应数据访问ConnectionFactory概念的API为PersistenceManagerFactory。在Spring

中，我们可以直接实例化特定的JDO实现产品提供的PersistenceManagerFactory 实现类，或者通过org.springframework.orm.jdo.LocalPersistenceManagerFactoryBean来配置和创建PersistenceManagerFactory。

使用JDO产品实现类直接创建PersistenceManagerFactory实例。要使用DataSource，我们可以直接实例化Jakarta Commons DBCP的BasicDataSource，也可以直接实例化C3P0的ComboPooledDataSource。对于PersistenceManagerFactory来说，我们也可以像实例化DataSource一样，直接实例化相应JDO产品提供的PersistenceManagerFactory实现类。代码清单15-21给出了在IoC容器中注册开源JDO实现JPOX的PersistenceManagerFactory实现类的代码示例。

代码清单15-21　IoC容器中实例化相应的PersistenceManagerFactory实现类代码示例

```
<bean id="mainDataSource" class="org.apache.commons.dbcp.BasicDataSource"➥
destroy-method="close">
    <property name="url">
        <value>${db.main.url}</value>
    </property>
    <property name="driverClassName">
        <value>${db.main.driver}</value>
    </property>
    <property name="username">
        <value>${db.main.user}</value>
    </property>
    <property name="password">
        <value>${db.main.password}</value>
    </property>
    ...
</bean>

<bean id="persistenceManagerFactory" ➥
class="org.jpox.PersistenceManagerFactoryImpl" destroy-method="close">
    <property name="connectionFactory" ref="mainDataSource"/>
    <property name="nontransactionalRead" value="true"/>
</bean>
```

当然，如果当前的应用程序不使用IoC容器，直接编程实例化即可。

通过LocalPersistenceManagerFactoryBean创建PersistenceManagerFactory实例。为便于在容器中配置和管理PersistenceManagerFactory，Spring提供了针对PersistenceManagerFactory的FactoryBean实现LocalPersistenceManagerFactoryBean。通过它，我们可以在容器中创建PersistenceManagerFactory实例并在整个的容器上下文中共享，例如：

```
<bean id="persistenceManagerFactory" ➥
class="org.springframework.orm.jdo.LocalPersistenceManagerFactoryBean">
    <property name="configLocation" value="classpath:jdo-config.properties"/>
</bean>
```

如果愿意，可以将该实例注入容器中任何依赖于PersistenceManagerFactory的bean定义。

创建完成的PersistenceManagerFactory可以注入Spring为JDO提供的模板方法类org.springframework.orm.jdo.JdoTemplate中，之后，所有的数据访问操作直接经由JdoTemplate进行。

JdoTemplate就是Spring为JDO提供的模板方法类，用来统一管理数据访问资源、异常处理以及事务控制。它的核心模板方法execute(JdoCallback)通过org.springframework.orm.jdo.JdoCall-

back回调接口提供具体的数据访问逻辑。JdoCallback接口的定义如下：

```
Object doInJdo(javax.jdo.PersistenceManager pm)
```

该回调接口为开发人员公开了PersistenceManager以进行基于JDO的数据访问操作。

JdoTemplate除了提供了该核心模板方法之外，为了简化较为普遍的数据访问操作，同时在核心模板方法基础上，提供了多组便利的模板方法。所有这些模板方法定义，都可以通过org.springframework.orm.jdo.JdoOperations接口定义获得。因为JdoTemplate本身就是实现了该接口以便为开发人员提供统一的数据访问接口。

2. Spring的JDO异常转译

如果直接使用JdoTemplate，那么有关JDOException到Spring异常体系的转译将会由JdoTemplate处理，我们无需关心特定于JDO的异常转译。不过，即使直接使用JDO的API进行数据访问（没有发现现在各个ORM产品在处理资源管理方面的API设计都改进很多吗），通过org.springframework.orm.jdo.PersistenceManagerFactoryUtils工具类提供的静态方法convertJdoAccessException()，我们依然可以获得从JDOException到Spring统一异常层次体系的转译支持，该方法如下所示：

```
public static DataAccessException convertJdoAccessException(JDOException ex)
```

另外，我们也可以通过PersistenceManagerFactoryUtils的newJdbcExceptionTranslator方法获得SQLExceptionTranslator的相应实例，来处理数据访问期间更加具体的SQLException，如下所示：

```
SQLExceptionTranslator sqlExTranslator = PersistenceManagerFactoryUtils. ➥
newJdbcExceptionTranslator(dataSource);
if(ex.getCause() instanceof SQLException)
    return sqlExTranslator.translate("task","sql",ex);
...
```

3. JdoDaoSupport

Spring为所有基于JDO的DAO实现类提供了org.springframework.orm.jdo.support.JdoDaoSupport作为继承的基类，所有继承了JdoDaoSupport的DAO实现类将直接获得JdoTemplate的数据访问支持，例如：

```
public class FooJdoDaoImpl extends JdoDaoSupport implements IFooDao
{
    public void delete(Object po)
    {
        getJdoTemplate().deletePersistent(po);
    }
    ...
}
```

在每个DAO使用前，我们需要为它注入一个JdoTemplate实例或者一个PersistenceManagerFactory实例。如果注入的是后者，JdoDaoSupport内部将根据提供的PersistenceManagerFactory构建相应的JdoTemplate使用。

使用JdoDaoSupport的好处是，整个团队可以以统一的DAO基类作为基础进行开发。JdoDaoSupport本身已经提供了最基本的基于JDO的数据访问支持，而且，如果愿意，我们也可以直接使用基类提供的PersistenceManagerFactory来使用JDO的原始API。当然，大多数情况下这是没有必要的。

如果系统已经存在DAO基类，那么扩展JdoDaoSupport显然是不现实的。这时我们可以直接为DAO注入JdoTemplate，或者重新设立一套Dao实现体系。但是，对于后者来说，往往是比较危险的举动。毕竟，一个团队内部的开发需要保持一致性，重新设立一套实现体系在技术层次上是比较合适的，但从管理和维护的角度，则确实有值得商榷的地方。

15.3.2　Spring对TopLink的集成

TopLink的命运比较曲折，作为最初由Object People公司开发的几乎是Java界ORM产品鼻祖的Toplink，在20世纪90年代末被短命的WebGain[①]收购之后，没过多久又被Oracle收购，并一直跟随Oracle走到今天，然后在2007年被Oracle开源。但不管怎么说，作为一个商业产品，Toplink还是值得世人称道的。

1. Spring中的Toplink资源管理

Toplink中没有明确的实现类对应数据访问ConnectionFactory的概念，所以，为了能够以统一的方式集成Toplink，Spring对Toplink中的ServerSession进行了进一步的抽象，使用org.springframework.orm.toplink.SessionFactory作为Spring中Toplink数据访问所对应的数据访问ConnectionFactory。

Spring通过org.springframework.orm.toplink.LocalSessionFactoryBean这个FactoryBean对org.springframework.orm.toplink.SessionFactory的配置和创建进行封装，这样，我们就可以在容器中添加如代码清单15-22所示的bean定义。

代码清单15-22　LocalSessionFactoryBean配置代码示例

```
<bean id="mainDataSource" class="org.apache.commons.dbcp.BasicDataSource" ➥
destroy-method="close">
    <property name="url">
        <value>${db.main.url}</value>
    </property>
    <property name="driverClassName">
        <value>${db.main.driver}</value>
    </property>
    <property name="username">
        <value>${db.main.user}</value>
    </property>
    <property name="password">
        <value>${db.main.password}</value>
    </property>
    ...
</bean>

<bean id="mySessionFactory" class="org.springframework.orm.toplink.LocalSessionFactoryBean">
    <property name="configLocation" value="toplink-sessions-config.xml"/>
    <property name="dataSource" ref="mainDataSource"/>
</bean>
```

可以通过编程的方式来实例化相应的org.springframework.orm.toplink.SessionFactory，也可以直接使用org.springframework.orm.toplink.LocalSessionFactory，LocalSessionFactoryBean是为IoC容器准备的，LocalSessionFactory则更多是为直接编程实例化准备的。

---

① 想当年WebGain开发的Java IDE也说得上“眉清目秀”惹人爱啊，个人感觉比IBM的VisualAge For Java要强多了，只不过，用短命来形容WebGain实在是太过贴切，呜呼，只能一声长叹。

org.springframework.orm.toplink.TopLinkTemplate是Spring为基于Toplink的数据访问提供的模板方法类，其核心模板方法定义如下：

```
public Object execute(TopLinkCallback action) throws DataAccessException
```

org.springframework.orm.toplink.TopLinkCallback是该核心模板方法所使用的回调接口，开发人员可以通过该回调接口提供具体的数据访问逻辑实现。TopLinkCallback接口定义如下：

```
public interface TopLinkCallback {
    Object doInTopLink(Session session) throws TopLinkException;
}
```

TopLinkCallback为开发人员公开了Session资源，供数据访问的时候使用，而对于资源的获取和释放则由TopLinkTemplate统一管理。

按照惯例，TopLinkTemplate在提供核心模板方法以满足一般数据访问需求的情况下，还会在该核心模板方法的基础上，提供更多便于基本数据访问操作的模板方法。所有这些模板方法全部由org.springframework.orm.toplink.TopLinkOperations接口来定义，TopLinkTemplate实现了该接口，所以提供了所有这些模板方法的实现供我们使用。

2. Toplink数据访问异常到Spring异常体系的转译

如果直接使用TopLinkTemplate进行数据访问，那么我们无需关心特定的数据访问异常，比如SQLException以及TopLinkException到Spring异常体系的转译，因为TopLinkTemplate内部已经帮助我们做了这些工作。

即使我们不通过TopLinkTemplate进行基于Toplink的数据访问，而是直接使用Toplink的原始API进行数据访问，依然能够获得TopLinkException到Spring异常体系的转译支持，只需要在处理异常的时候通过org.springframework.orm.toplink.SessionFactoryUtils的静态方法convertTopLinkAccessException进行异常转译即可，如下：

```
public static DataAccessException convertTopLinkAccessException(TopLinkException ex)
```

当然，无论如何，通过TopLinkTemplate进行基于Toplink的数据访问才是比较合适的方式。

3. TopLinkDaoSupport

Spring为使用基于Toplink的数据访问技术实现的所有DAO，提供了org.springframework.orm.toplink.support.TopLinkDaoSupport作为整个DAO实现体系的顶层基类，所有基于Toplink的数据访问DAO实现类，直接继承TopLinkDaoSupport就可获得TopLinkTemplate或者Toplink数据访问API的支持，例如：

```
public class FooToplinkDaoImpl extends TopLinkDaoSupport implements IFooDao{
    public Object find(String pk) {
        return getTopLinkTemplate().readById(Foo.class,pk);
    }
    ...
}
```

除了提供TopLinkTemplate以进行基于Toplink的数据访问，我们还可以通过TopLinkDaoSupport获取Toplink的原始API进行数据访问，同时，TopLinkDaoSupport还提供了相应的异常转译方法，进一步提升了TopLinkDaoSupport作为DAO基类的存在价值。

15

15.3.3 Spring 对 JPA 的集成

JPA（Java Persistence API，Java持久化API）是Sun于Java EE 5之后提出的ORM解决方案的统一标

准，具体实现由不同提供商提供，包括Hibernate、Toplink等。就好像当年的JDBC标准一样，只不过，JPA是面向ORM的统一。

Spring框架在2.0版本之后提供了对JPA的支持，并且在之后的版本中做进一步的统一和完善。

1. Spring中JPA的资源管理

JPA中的EntityManagerFactory对应数据访问的ConnectionFactory的概念，Spring提供了如下三种方式来配置和获取EntityManagerFactory。

使用LocalEntityManagerFactoryBean。对于基于Java SE的独立应用程序或者测试环境下的开发，我们可以使用org.springframework.orm.jpa.LocalEntityManagerFactoryBean来获取相应的EntityManagerFactory。LocalEntityManagerFactoryBean将默认读取META-INF/persistence.xml配置文件信息，来构建相应的EntityManagerFactory，我们只需要指定相应的persistenceUnitName即可，如下所示：

```
<bean id="entityManagerFactory"
    class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
    <property name="persistenceUnitName" value="yourPersistenceUnitName"/>
</bean>
```

这种方式应用场景有限，对于更多定制需求，可以考虑使用LocalContainerEntityManagerFactoryBean。

使用LocalContainerEntityManagerFactoryBean。org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean是比较常用的配置和获取EntityManagerFactory的方式。通过它，我们可以指定自定义的配置文件位置、独立的dataSource定义甚至Spring提供的定制字节码转换的loadTimeWeaver实现。代码清单15-23给出了Spring IoC容器内LocalContainerEntityManagerFactoryBean通常的配置代码示例。

代码清单15-23　LocalContainerEntityManagerFactoryBean配置代码示例

```
<bean id="mainDataSource" class="org.apache.commons.dbcp.BasicDataSource"➥
destroy-method="close">
    <property name="url">
        <value>${db.main.url}</value>
    </property>
    <property name="driverClassName">
        <value>${db.main.driver}</value>
    </property>
    <property name="username">
        <value>${db.main.user}</value>
    </property>
    <property name="password">
        <value>${db.main.password}</value>
    </property>
    ...
</bean>

<bean id="entityManagerFactory" class="org.springframework.orm.jpa. ➥
LocalContainerEntityManagerFactoryBean">
    <property name="dataSource" ref="mainDataSource"/>
    <property name="loadTimeWeaver">
        <bean class="org.springframework.instrument.classloading. ➥
        InstrumentationLoadTimeWeaver"/>
    </property>
```

```
        <property name="persistenceXmlLocation" value="META-INF/main-persistence.xml"/>
    </bean>
```

通过JNDI获取**EntityManagerFactory**。我们也可以使用绑定到JNDI的EntityManager-Factory，这对基于Spring的IOC容器的应用来说，也只是配置文件的少许改动，例如：

```
<bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName">
        <value>persistence/pUnitName</value>
    </property>
</bean>
```

当然，Spring 2.0之后，我们更喜欢用XSD风格的配置，如下所示：

```
<jee:jndi-lookup id="entityManagerFactory" jndi-name="persistence/pUnitName"/>
```

相对来说，这种方式更加简洁，描述性也更强。

获取EntityManagerFactory实例之后，我们就可以使用Spring为JPA提供的模板方法类org.springframework.orm.jpa.JpaTemplate进行数据访问了。JpaTemplate的核心模板方法为Object execute(JpaCallback action)，开发人员直接通过org.springframework.orm.jpa.JpaCallback回调接口提供具体的数据访问逻辑即可。资源管理、事务以及异常处理等关注点，则由JpaTemplate模板方法来统一管理。

JpaCallback接口定义为开发人员公开了EntityManager实例，用于具体的基于JPA的数据访问操作，该接口定义如下：

```
public interface JpaCallback {
    Object doInJpa(EntityManager em) throws PersistenceException;
}
```

通过EntityManager，开发人员可以完成任何基于JPA原始API可以完成的工作，而EntityManager获取以及释放的管理问题，则由JpaTemplate来处理。

org.springframework.orm.jpa.JpaOperations接口定义了所有JpaTemplate中可用的数据访问模板方法，除了核心的execute(JpaCallback)模板方法，还有一些便于常用操作的模板方法，具体可以参阅JpaOperations或者JpaTemplate的Javadoc文档。

> **注意** 更多有关JpaTemplate使用以及EntityManagerFactory实例配置和获取的信息，请参照Spring 2.0之后提供的参考文档。

2. Spring中JPA的异常转译

JpaTemplate模板方法内部同时处理了JPA的数据访问异常到Spring统一异常体系的转译。如果使用JpaTemplate进行数据访问，那么不用关心对于特定于JPA的数据访问异常的处理。

即使我们在自己的DAO实现中没有使用JpaTemplate，而是使用JPA的原始API进行数据访问，依然能够得到Spring统一异常体系的好处。实际上，JpaTemplate内部的异常转译最终是通过org.springframework.orm.jpa.EntityManagerFactoryUtils提供的静态方法convertJpaAccess-ExceptionIfPossible()完成的，该方法签名定义如下：


```
public static DataAccessException convertJpaAccessExceptionIfPossible(RuntimeException ex)
```

如果需要，我们同样可以直接使用该工具类提供的这个方法来完成到Spring统一异常体系的转译。

3. `JpaDaoSupport`

Spring为所有基于JPA进行数据访问的DAO实现类提供了一个统一的基类，即`org.springframework.orm.jpa.support.JpaDaoSupport`，相应的DAO实现类可以直接继承`JpaDaoSupport`以获得JpaTemplate提供的JPA数据访问支持。

当然，`JpaDaoSupport`允许我们注入`EntityManagerFactory`或者`EntityManager`实例。如果愿意，我们也可以直接使用JPA的原始API进行数据访问。

> **注意** 本章的主要内容是Spring通过什么理念和操作方式对各种数据访问进行集成，所以，并没有对各种数据访问技术进行详尽的介绍。实际上，要发挥各种数据访问技术的最大功效，我们依然需要精通使用的数据访问技术才行，而这与集成没有太多关系的。集成只是让事情变得更简单。

15.4 小结

Spring框架为当下Java平台上的各种ORM解决方案提供了统一的集成支持，包括数据访问资源的模板化管理、特定的数据访问异常到Spring异常层次体系的转译，以及稍后将会提到的Spring事务管理支持等。

本章主要针对Hibernate和iBATIS两种目前比较流行的ORM解决方案的Spring集成进行了比较详尽的介绍。鉴于Spring对各种ORM产品的集成一脉相承，对其他ORM产品的集成则顺带提及。在阅读完本章内容后，你应该已经对生活在Spring数据访问层的各种ORM产品的使用了如指掌了。

第16章 Spring数据访问之扩展篇

本章内容

- 活用模板方法模式及Callback
- 数据访问中的多数据源
- Spring 3.0展望

16.1 活用模板方法模式及 Callback

纵观第13章到第15章的内容，我们会发现不管是Spring对JDBC API的抽象，还是对Hibernate、iBATIS等ORM的集成，全部都采用了一种理念或者处理方式，那就是模板方法模式与相应的Callback接口相结合。

那么，为什么要在这里使用模板方法与Callback相结合的问题处理方式呢？最基本的原因是：不管是JDBC还是Hibernate或者其他ORM实现，在资源管理上有一个共性，那就是需要在资源使用之后可以安全地释放这些资源。与*Bitter Java*所提出的理念相同，为了确保尽可能地将资源的获取和资源的释放操作放在一起，Spring在数据访问层处理资源的问题上，采用了模板方法模式。这样，以一种统一而集中的方式来处理资源的获取和释放，避免了将这种容易出现问题的操作分散到代码中的各个地方，进而也就避免了由此产生的“资源泄漏”一类比较严重的问题。

推而广之，我们可以以相同的模式来处理类似的问题，而也会发现，这样的处理与我们之前的处理或者封装方式是如此的不同，如此的简洁明了。

16.1.1 FTPClientTemplate

之前我们说过，通常的FX系统会从相应的新闻提供商那里定期获取外汇交易相关新闻。最常见的方式就是，通过FTP协议到指定的FTP服务器去定期下载相应的新闻文件，所以，FX系统的应用程序需要提供相应的实现类来进行FTP操作。而程序中的FTP操作应该是比较通用的，无非就是上传下载文件。为了程序能有一个良好的结构，我们通常会将这些FTP操作逻辑封装为一个工具类。而下面我们将看到的，就是两种截然不同的工具类实现方式。

我们不需要为最为底层的FTP操作“重新发明轮子”，Jakarta Commons Net类库提供了基本的FTP支持，不过，直接使用Commons Net的API就与直接使用JDBC API一样让人尴尬，代码清单16-1的代码演示了直接使用Commons Net API进行FTP操作的一般情况。

代码清单16-1　FTPClient的Javadoc文档中的使用示例代码摘录

```
boolean error = false;
try {
```

```
    int reply;
    ftp.connect("ftp.foobar.com");
    System.out.println("Connected to " + server + ".");
    System.out.print(ftp.getReplyString());

    // After connection attempt, you should check the reply code to verify
    // success.
    reply = ftp.getReplyCode();

    if(!FTPReply.isPositiveCompletion(reply)) {
      ftp.disconnect();
      System.err.println("FTP server refused connection.");
      System.exit(1);
    }
    ... // transfer files
    ftp.logout();
  } catch(IOException e) {
    error = true;
    e.printStackTrace();
  } finally {
    if(ftp.isConnected()) {
      try {
        ftp.disconnect();
      } catch(IOException ioe) {
        // do nothing
      }
    }
    System.exit(error ? 1 : 0);
  }
```

我得承认，FTPClient类提供的这段代码只是一段示例，不能在实际的生产环境下使用，所以，我们尝试对其进行封装。

对于使用FTPClient类实现的FTP操作来说，无非就是登录FTP服务器，传输文件，然后退出服务器三步。代码清单16-2所示的FTP操作工具类是最为常见的实现方式。

代码清单16-2 初步的FTP操作封装工具类原型代码示例

```
class Phase1FtpUtility
{
     public boolean login(...)
     {
            ... // 登录代码
     }
     public void doTransfer(...)
     {
           ...// 文件传输逻辑实现
     }
     public boolean logout()
     {
            ...// 退出登录
     }
 }
```

相对于示例中的代码来说，通过Phase1FtpUtility类的封装，现在进行FTP操作看起来要简洁多了。不过，这样的封装方式并没有起到多少实际效果。

Phase1FtpUtility对FTPClient API的封装力度不够，与直接使用FTPClient的API相比，调用方只是少写几行代码而已，如下所示：

```
Phase1FtpUtility ftpUtility = ...;

if(ftpUtility.login(..))
```

```
{
    ftpUtility.doTransfer(..);
}

ftpUtiligy.logout();
```

而且，这样的代码把资源的管理留给了每处调用Phase1FtpUtility进行FTP操作的调用代码。与数据库连接一样，你要怎样保证相应的资源在每处都能成功地获得释放呢？就现有的API封装方式，我们只能加强开发人员的约束力来达到正确使用API的目的了。

通常，Phase1FtpUtility的doTransfer(..)方法用来实现具体的FTP操作逻辑，但现在的Phase1FtpUtility只能提供固定的FTP操作逻辑。如果其他调用方需要不同的FTP操作，那么，或许就得子类化Phase1FtpUtility并覆写doTransfer(..)方法了。不过，这样好像偏离了我们要将Phase1FtpUtility作为单一工具类来使用的初衷。

鉴于这些限制，我们需要另一种FTPClient API的封装方式。而你也看出来了，就如Phase1FtpUtility所展示的那样，所有的使用FTPClient进行FTP操作的步骤几乎是一样的，唯一的不同就是每次进行FTP操作的细节。在经过JdbcTemplate、HibernateTemplate以及SqlMapClientTemplate等熏陶之后，自然而然就应该想到，我们可以对FTPClient API进行同样的处理，从而代码清单16-3所给出的FTPClientTemplate就是我们的最终需要。

代码清单16-3 FTPClientTemplate定义

```
public class FTPClientTemplate {
    //
    private static final Log logger = LogFactory.getLog(FTPClientTemplate.class);
    //
    private FTPClientConfig ftpClientConfig;        // 可选属性
    //
    private String server;                          // 必须指定
    private String username;                        // 必须指定
    private String password;                        // 必须指定
    private int    port=21;                         // 必须指定


    public FTPClientTemplate(String host,String username,String password)
    {
        this.server   = host;
        this.username = username;
        this.password = password;
    }

    public void execute(FTPClientCallback callback) throws IOException
    {
        FTPClient ftp = new FTPClient();
        try {
            if(this.getFtpClientConfig() != null)
            ftp.configure(this.getFtpClientConfig());

            ftp.connect(server,getPort());
            // 检查到服务器的连接是否正确
            int reply = ftp.getReplyCode();
            if(!FTPReply.isPositiveCompletion(reply))
            {
                throw new IOException("failed to connect to the FTP Server:"+server);
            }
            // 登录
            boolean isLoginSuc=
                ftp.login(this.getUsername(),this.getPassword());
```

16

```
            if(!isLoginSuc)
            {
                throw new IOException("wrong username or password,"+
                "please try to login again.");
            }
            // 通过回调方法实现特定的FTP操作
            callback.processFTPRequest(ftp);

            // 退出登录
            ftp.logout();
        }
        finally
        {
            if(ftp.isConnected())
            {
                ftp.disconnect();
            }
        }
    }

    public String[] listFileNames(final String remoteDir,final String fileNamePattern) ➥
    throws IOException
    {
    final List<String[]> container = new ArrayList<String[]>();

    execute(new FTPClientCallback(){
        public void processFTPRequest(FTPClient ftp) throws IOException {
            ftp.enterLocalPassiveMode();
            changeWorkingDir(ftp,remoteDir);

            if(logger.isDebugEnabled())
                logger.debug("working dir:"+ ftp.printWorkingDirectory());
            }});
    return container.get(0);
    }

    protected void changeWorkingDir(FTPClient ftp, String remoteDir) throws IOException {
        Validate.notEmpty(remoteDir);
        ftp.changeWorkingDirectory(remoteDir);
    }
    ...

    // setter和getter方法定义
    ...
}
```

我们通过execute(FTPClientCallback)方法对整个的基于FTPClient的API使用流程进行了封装，而将我们真正关心的每次具体的FTP操作交给了FTPClientCallback，该接口定义如下所示：

```
public interface FTPClientCallback {
    public void processFTPRequest(FTPClient ftpClient) throws IOException;
}
```

现在要做的，就是根据每次FTP操作请求细节提供相应的FTPClientCallback实现，然后交给FTPClientTemplate执行，如下所示：

```
FTPClientTemplate ftpTemplate = new FTPClientTemplate(host,user,pwd);
FTPClientCallback callback = new FTPClientCallback(){
    public void processFTPRequest(FTPClient ftpClient) throws IOException
    {
        ... // 特定的FTP操作实现逻辑
    }
```

```
};
ftpTemplate.execute(callback);
```

FTPClientTemplate一旦构建完成，其他任何调用方都可以共享使用它，调用方每次提供自己的FTPClientCallback实现即可。

现在的FTPClientTemplate看起来可能过于单薄。某些常用的FTP操作，如文件上传、文件下载、文件列表读取等，我们可以在FTPClientTemplate内直接提供，而没有必要让调用方每次实现几乎相同的代码。listFileNames(remoteDir,fileNamePattern)方法就是这样的方法实现，无非就是提供了相应的FTPClientCallback实现，然后最终委托execute()方法执行而已。

作为工具类，我们可以直接将这些常用的FTP操作方法定义到FTPClientTemplate中。如果愿意，也可以设计一个FTPOperation之类的接口，里面定义一系列的FTP操作方法，然后让FTPClientTemplate来实现该接口。

16.1.2 HttpClientTemplate

现在基于REST方式的Web 服务好像比原来SOAP的方式更加受人欢迎一些，世界上许多券商或者银行通常也会以REST的方式发送一些外汇牌价之类的信息，甚至，FX系统的某些外汇新闻提供商也通过HTTP协议采用类似于REST的方式来发送新闻。那么，与基于FTP协议的信息交换类似，对于这种方式的信息交换，我们也需要在应用程序中采用适当的API进行处理。

Apache Commons HttpClient是一个提供HTTP协议支持的Java类库，许多应用包括稍后我们将提到的Spring Remoting都是采用该类库实现的。我们同样可以使用该类库进行基于HTTP协议的信息交换，或者说得更“时髦”一点儿，进行REST方式的Web服务开发。

如果你是初次接触HttpClient，那么应该先看一下HttpClient网站提供的Tutorial文档，里面给出了类似代码清单16-4给出的使用代码示例。

代码清单16-4 HttpClient的一般使用代码示例

```
public class HttpClientTutorial {

  private static String url = "http://www.apache.org/";

  public static void main(String[] args) {
    // Create an instance of HttpClient.
    HttpClient client = new HttpClient();

    // Create a method instance.
    GetMethod method = new GetMethod(url);

    // Provide custom retry handler is necessary
    method.getParams().setParameter(HttpMethodParams.RETRY_HANDLER,
            new DefaultHttpMethodRetryHandler(3, false));

    try {
      // Execute the method.
      int statusCode = client.executeMethod(method);

      if (statusCode != HttpStatus.SC_OK) {
        System.err.println("Method failed: " + method.getStatusLine());
      }

      // Read the response body.
      byte[] responseBody = method.getResponseBody();

      // Deal with the response.
```

```
            // Use caution: ensure correct character encoding and is not binary data
            System.out.println(new String(responseBody));

        } catch (HttpException e) {
            System.err.println("Fatal protocol violation: " + e.getMessage());
            e.printStackTrace();
        } catch (IOException e) {
            System.err.println("Fatal transport error: " + e.getMessage());
            e.printStackTrace();
        } finally {
            // Release the connection.
            method.releaseConnection();
        }
    }
}
```

又是获取资源、操作、释放资源、处理异常等，而且这样的代码无法用于生产环境也是肯定的了，那该怎么处理，我想你已经心里有数了吧？余下的部分，还是留给你来实现吧。

16.2 数据访问中的多数据源

在实际的项目中，我们很少只使用一个数据库。出于灾难恢复或者负载均衡之类目的考虑，生产环境中通常都会存在多台数据库服务器，相应地，在应用程序对这些数据库进行数据访问的时候，我们通常会碰到一个比较常见的问题，即如何管理数据访问过程中涉及的多个数据源。

下面我们不妨从两个角度来阐述一下在应用程序中如何对多个数据源进行管理。

16.2.1 “主权独立”的多数据源

所谓“主权独立”是指系统中的每个数据源都对外独立承担公开数据库资源的职能，如图16-1所示。

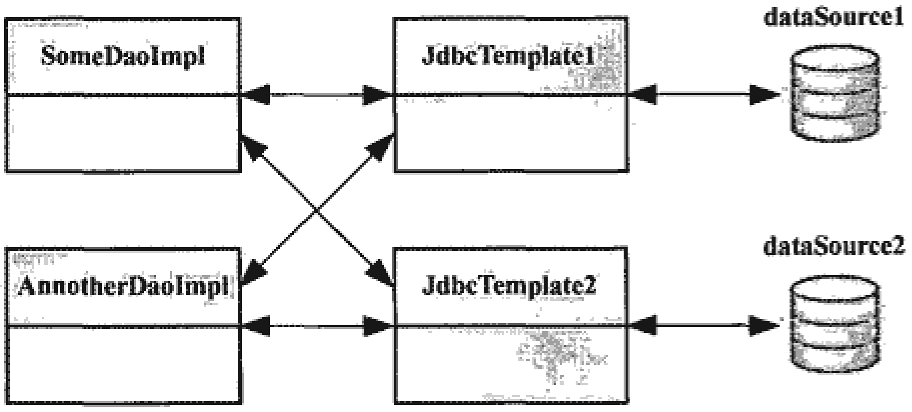

图16-1 “主权独立”的多数据源应用场景示意图

具体的应用场景如下。

- 每个数据库所存储的数据性质不同，比如数据库A存储重要的交易信息，数据库B存储次要的系统管理信息等。如果要访问交易信息，那么通常可以明确指定使用对应数据库A的dataSourceA进行数据访问。如果要访问系统管理信息，则明确指定使用对应数据库B的dataSourceB，依此类推。
- 每个数据库分别承担不同的数据访问请求形式，比如数据库A只允许更新操作不允许查询，数据库B只允许查询不允许更新等，这时，也是可以明确指定使用哪个dataSource进行数据访问。

当然，场景并非只有这些，但总的意图是相似的，那就是每个dataSource的职能对于使用它们的客户端来说足够明确，完全是各自独立使用。

我参与开发的FX项目中就使用了这样的多数据源管理方式。在FX中，设置的MAIN数据库主要存储顾客或者银行与FX Broker之间的交易信息，设置的INFO数据库主要存储汇率以及系统履历之类的信息。通常情况下，交易以及汇率之间的信息存储，在逻辑上是可以分开进行的，所以现在，应用程序对应的Spring配置内容基本如代码清单16-5所示。

代码清单16-5 主权独立的多数据源在Spring中的配置示例

```
<bean id="mainDataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method ➥
="close">
    <property name="url" value="${main.jdbcUrl}"/>
    <property name="driverClassName" value="${main.driver}"/>
    <property name="username" value="${main.username}"/>
    <property name="password" value="${main.password}"/>
    <!-- other property settings -->
</bean>

<bean id="infoDataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method ➥
="close">
    <property name="url" value="${info.jdbcUrl}"/>
    <property name="driverClassName" value="${info.driver}"/>
    <property name="username" value="${info.username}"/>
    <property name="password" value="${info.password}"/>
    <!-- 其他属性设置 -->
</bean>

<bean id="mainJdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
    <property name="dataSource" ref="mainDataSource"/>
</bean>

<bean id="infoJdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
    <property name="dataSource" ref="infoDataSource"/>
</bean>

<bean id="dataAccessResourceSupport" abstract="true">
    <property name="mainJdbcTemplate" ref="mainJdbcTemplate"/>
    <property name="infoJdbcTemplate" ref="infoJdbcTemplate"/>
</bean>

<bean id="someDaoWithMainDS" class="...">
    <property name="mainJdbcTemplate" ref="mainJdbcTemplate"/>
    <!-- 其他属性设置 -->
</bean>

<bean id="someDaoWithInfoDS" class="...">
    <property name="infoJdbcTemplate" ref="infoJdbcTemplate"/>
    <!-- 其他属性设置 -->
</bean>

<bean id="someDaoWithBothDS" class="..." parent="dataAccessResourceSupport">
    <!-- 其他属性设置 -->
</bean>
```

这种情况下的多数据源管理是最简单的，也是比较容易管理的。所以，你在考虑下面将要介绍的这种更加动态、更加复杂的多数据源管理方式之前，请先对自己的数据访问场景做一个评估，看一下当前这种方式是否已经足够满足项目的数据访问需要，实在不行的话，再考虑后继方案，也就是在运行期间来决定到底使用多个数据源中的哪一个。

16

16.2.2　“合纵连横”的多数据源

社区中经常提到的多数据源互换即属于这种场景。之所以用“合纵连横”来形容这些数据源是因为，对于使用它们的数据访问类来说，这些数据源已经丧失了“独立自主”的地位，所有与数据访问类进行的交互需要通过“盟主”进行，该盟主本质上也是一个DataSource，但它的职责更加倾向于对“联盟”内的多个DataSource的职能进行协调和管理，最终数据访问所需要的资源由“盟主”来决定要哪一个DataSource提供（见图16-2）。

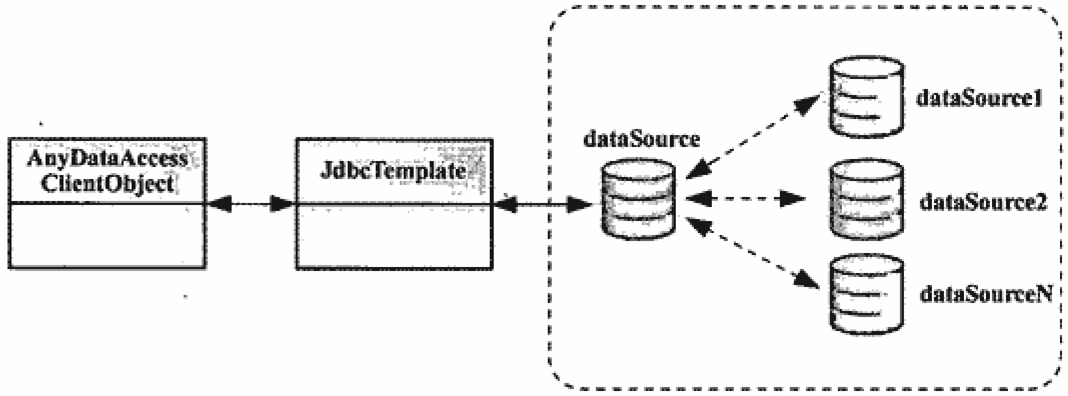

图16-2　“合纵连横”的多数据源应用场景示意图

使用这种多数据源管理方式的具体场景如下所述。

(1) 在系统中设置多台“地位相当”的数据库以实现多机热备，从而保证数据库的高可用性（HA，High Availability）。这时，如果某一台数据库挂掉，可以迅速切换到另一台数据库，而对于数据访问类来说，这样的切换是透明的。

(2) 系统中存在的多台服务器也是“地位相当”的。不过，同一时间它们都处于活动（Active）状态，出于负载均衡等因素考虑，数据访问请求需要在这几台数据库服务器之间进行合理分配。这时，通过统一的一个DataSource来屏蔽这种请求分配的需求，从而解除数据访问类与具体DataSource的耦合。

(3) 系统中存在的多台数据库服务器地位可能相当，也可能不相当，但数据访问类在系统启动时间无法明确到底应该使用哪个数据源进行数据访问，而必须在系统运行期间通过某种条件来判定到底应该使用哪个数据源。这时，我们也得使用这种“合纵连横”的方式向数据访问类公开一个统一的DataSource，由该DataSource来解除数据访问类与具体数据源之间的过紧耦合。

更多场景需要你根据具体的应用来判定。不过，并非所有的应用都要做这样的处理。如果能够保持简单，那么尽量保持简单，毕竟，我们提倡K.I.S.S.（Keep It Simple,Stupid）。

要实现这种“合纵连横”的多数据源管理方式，总的指导原则就是实现一个自定义的DataSource。让该DataSource来管理系统中存在的多个与具体数据库挂钩的数据源，数据访问类只跟这个自定义的DataSource打交道即可。在Spring 2.0.1发布之前，各个项目中可能存在多种针对这种情况的多数据源管理方式。Spring 2.0.1发布之后，引入了AbstractRoutingDataSource，使用该类可以实现普遍意义上的多数据源管理功能。

假设有三台数据库用来实现负载均衡，所有的数据访问请求最终需要平均的分配到这三台数据库服务器上，那么，可以通过继承AbstractRoutingDataSource来快速实现一个满足这样场景的原型(Prototype),见代码清单16-6。

代码清单16-6　管理多数据源负载均衡的自定义数据源原型代码示例

```
public class PrototypeLoadBalanceDataSource extends AbstractRoutingDataSource  {
```

```
    private Lock            lock = new ReentrantLock();
    private int          counter = 0;
    private int dataSourceNumber = 3;

    @Override
    protected Object determineCurrentLookupKey() {
        lock.lock();
        try
        {
            counter++;
            int lookupKey = counter % getDataSourceNumber();
            return new Integer(lookupKey);
        }
        finally
        {
            lock.unlock();
        }
    }
    // ...
}
```

我们在介绍AbstractRoutingDataSource的时候说过，要继承该类，通常只需要给出determineCurrentLookupKey()方法的逻辑即可。

代码清单16-7是针对PrototypeLoadBalanceDataSource的配置。

代码清单16-7 针对PrototypeLoadBalanceDataSource的配置代码示例

```
<bean id="dataSource1" class="org.apache.commons.dbcp.BasicDataSource" destroy-method ➥
="close">
    <property name="url" value=".."/>
    <property name="driverClassName" value=".."/>
    <property name="username" value=".."/>
    <property name="password" value=".."/>
    <!-- other property settings -->
</bean>

<bean id="dataSource2" class="org.apache.commons.dbcp.BasicDataSource" destroy-method ➥
="close">
    <property name="url" value=".."/>
    <property name="driverClassName" value=".."/>
    <property name="username" value=".."/>
    <property name="password" value=".."/>
    <!-- other property settings -->
</bean>

<bean id="dataSource3" class="org.apache.commons.dbcp.BasicDataSource" destroy-method ➥
="close">
    <property name="url" value=".."/>
    <property name="driverClassName" value=".."/>
    <property name="username" value=".."/>
    <property name="password" value=".."/>
    <!-- other property settings -->
</bean>

<util:map id="dataSources">
    <entry key="0" value-ref="dataSource1"/>
    <entry key="1" value-ref="dataSource2"/>
    <entry key="2" value-ref="dataSource3"/>
</util:map>

<bean id="dataSourceLookup" class="org.springframework.jdbc.datasource.lookup. ➥
  MapDataSourceLookup">
```

```
        <constructor-arg>
            <ref bean="dataSources"/>
        </constructor-arg>
    </bean>

    <bean id="dataSource" class="..PrototypeLoadBalanceDataSource">
        <property name="defaultTargetDataSource" ref="dataSourc1"/>
        <property name="targetDataSources" ref="dataSources"/>
        <property name="dataSourceLookup" ref="dataSourceLookup"/>
    </bean>

    <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <bean id="someDao" class="...">
        <property name="jdbcTemplate" ref="jdbcTemplate"/>
        <!-- 其他属性设置 -->
    </bean>
```

因为我们不想使用AbstractRoutingDataSource默认的键查找行为（根据指定的键通过JNDI进行查找），所以为PrototypeLoadBalanceDataSource重新设置了DataSourceLookup，转而使用MapDataSourceLookup。

提示　在PrototypeLoadBalanceDataSource中，我们直接将查找的键硬编码到了代码中。实际上，更多时候，我们会将键绑定到当前线程上，而在determineCurrentLookupKey()方法内直接从当前线程取得绑定的键返回即可。而这种情况下对键的更改也变得更加灵活多变，比如，我们可以在数据访问类内直接将要访问的数据源对应的查找键绑定到当前线程，也可以在系统的某个位置设置拦截器，当拦截到相应事件的时候，根据逻辑设置绑定到当前线程的查找键等。你可以先思考如果让你来实现这么一个AbstractRoutingDataSource，应该如何处理。不过，如果你已经迫不及待，那么可以直接转向Spring的事务管理章节中的扩展篇，我将在那里为你展示详细的实现过程。

有了AbstractRoutingDataSource之后，实现这种“合纵连横”的多数据源管理，将不再像最初看起来那么复杂和神秘。

16.2.3　结束语

因为以上两种多数据源的管理方式在实际的使用过程中可能还有一些变数，所以最后，还是应该提及如下两点注意事项。

(1) 不管是“独立主权”的多数据源管理方式还是“合纵连横”的多数据源管理方式，单独使用任何一种都是有其特定的应用场景的。不过，这并不意味着二者是相互竞争甚至割裂的。如果必要，我们完全可以组合两种多数据源管理方式。

如果将“合纵连横”的多数据源作为一个整体放入“独立主权”的多数据源场景中，我们可以得到如图16-3所示的一幅画卷。

如果将“独立主权”的多个数据源先分别注入上一层对象，然后将上一层对象和数据源作为一个整体再并入“合纵连横”的多数据源场景，那么，我们又可以得出另一幅图景（如图16-4所示）。

当然，如果愿意，我们还可以根据情况采取进一步的组合措施。不过，在进行之前，还是需要先全面评估一下整体情况，看是否真的需要这么做。毕竟，复杂度的过多引入有些时候并非必要。

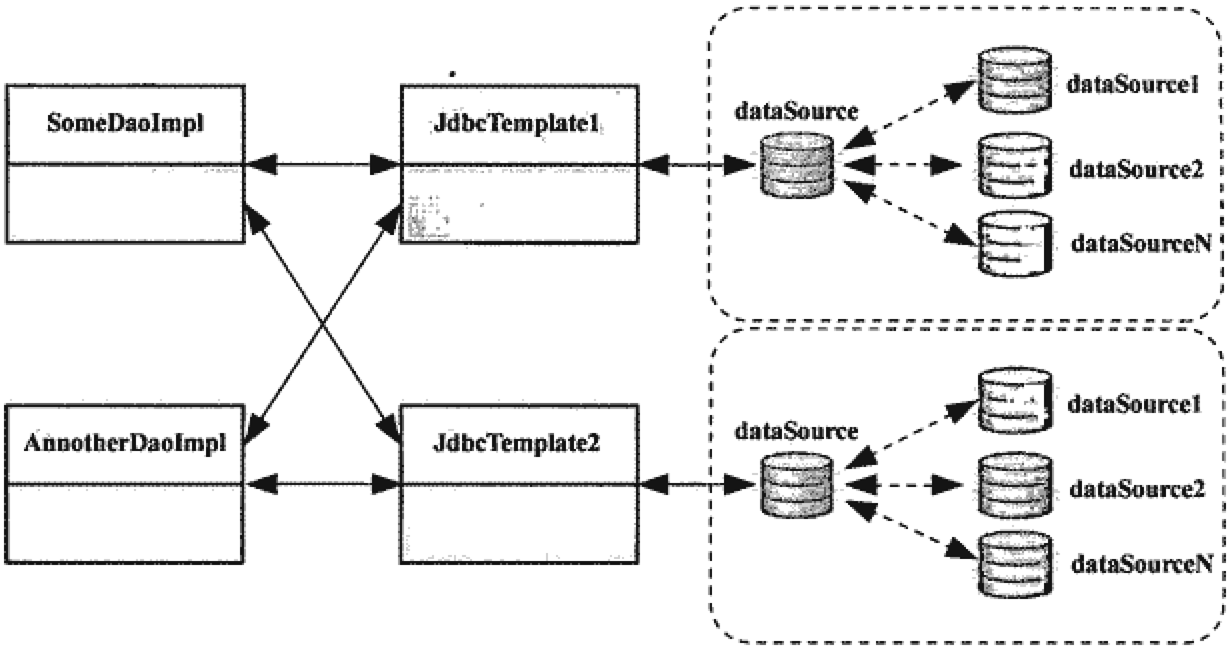

图16-3 “合纵连横”争取“主权独立”

(2) 在“主权独立”的多数据源场景中，我们是将独立的数据源注入给了JdbcTemplate，但这只是为了演示的目的，实际上，对于iBATIS和Hibernate来说，这样的场景也是类似的。

不过，在“合纵连横”的多数据源场景中，将JdbcTemplate的使用类推iBATIS，即SqlMap-ClientTemplate是可以的。但以同样的方式类推到Hibernate则有需要注意的地方。我们可以将“合纵连横”的多个数据源注入给Hibernate的SessionFactory（实际上是通过Spring的LocalSession-FactoryBean），然后HibernateTemplate直接引用这个SessionFactory即可。但当开启了Hibernate的二级缓存的时候（与SessionFactory挂钩），这样的多个数据源直接注入SessionFactory并且可以动态查找替换的方式可能造成问题。如果二级缓存中有与当前使用的DataSource挂钩的内容，而这时切换到了下一个DataSource，那么二级缓存里的内容需要根据情况进行合理的处理，或者清空，或者通过某种方式来同步，否则在并发的情况下，难免出现问题。当然，如果可以忽略这样的数据冲突，那么可能也有不处理的理由。

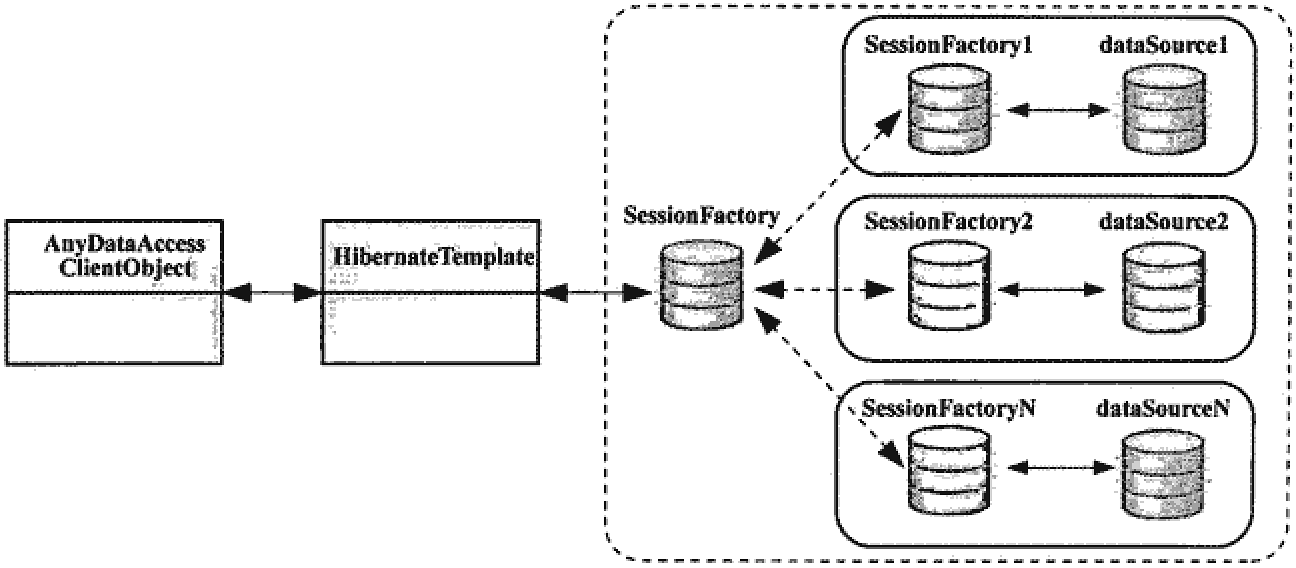

图16-4 从“主权独立”并入“合纵连横”

16

如果不需要开启Hibernate的二级缓存，或者可以忽略二级缓存数据的不一致性，那么，采用“合纵连横”的多数据源直接注入SessionFactory的方式来实现多个数据源的管理是可以的。否则，可以将“独立主权”的多数据源管理方式并入“合纵连横”的多数据源管理方式，以SessionFactory一级替代DataSource一级，这也就是第二种组合场景所描绘的那样。这时，我们可以像Spring提供AbstractRoutingDataSource那样，提供一个AbstractRoutingSessionFactory，也可以自己实现一个SessionFactory来屏蔽多个具体的SessionFactory。总之方法和原则与多个数据源的处理方式是类似的，至于要采用设计模式还是AOP，那就看你了。

最后，祝各位在多数据源管理的道路上一帆风顺，即使不顺，那么应用以上的方式“排除万难去争取胜利”也不再是难事了吧？

16.3 Spring 3.0 展望

在Spring 3.0将全面升级到Java 5这一前提之下，我们不难想象，像JdbcTemplate这样的重量级选手绝对会被“重新塑造”一番。虽然SimpleJdbcTemplate之前已经有“担此重任”之势，但既然Spring 3.0中的API将大修一把，SimpleJdbcTemplate可能还会保持，但只可能局限在某些场景下的使用。更多时候，泛型化之后的JdbcTemplate依然会是Spring数据访问层的主力战将。

虽然Spring的数据访问层提供了两种数据访问最佳实践方式，但是，基于操作对象的实践方式在实际使用过程中并非像预想的那样受欢迎，除了StoredProcedure确切地可以减轻存储过程调用的复杂度，其他类型的操作对象完全可以被JdbcTemplate轻松替代。基于这一前提，在Spring 3.0中将更加提倡基于JdbcTemplate的数据访问实践方式，除了StoredProcedure，其他类型的操作对象将不推荐使用。

随着JPA规范的不断推行，各种ORM解决方案之间的差异性将不断被弱化，而且，JPA 2.0规范也在紧锣密鼓的制定中。不难想象，Spring 3.0为ORM提供集成支持的时候，将更多的向JPA倾斜。正像Spring团队在自己的博客中所说，他们将尽早地提供对JPA 2.0的支持，这更使得JPA的排名预期持续升高。另外，因为Toplink一直没有引起更多的目光关注，预计Spring 3.0中将放弃对Toplink的集成支持。

16.4 小结

Spring的数据访问层对某些问题的处理方式和最佳实践是值得我们借鉴的。本章我们引申了Spring数据访问层中重用模板方法模式与Callback接口相结合的问题处理的理念，借助于FTPClient和HttpClient的示例，帮助大家思考。希望大家可以举一反三，在日常开发中的合适场景采用Spring数据访问层中重用的这种实践方式。

另外，本章也简单分析和阐述了数据访问中涉及的多数据源管理的问题，帮助大家理清思路，以便大家在日常开发工作中得心应手的处理类似问题。

Part 5 第五部分

事务管理

事务管理（Transaction Management）是一个很深的研究方向，而本部分的最终目的是阐述Spring事务管理抽象层的理念以及相关内容。你大可放心，我不会为大家准备一本类似于“砖头”的书。为了能够在整个讲解的过程中有一个平滑的过渡，有关事务（Transaction）的一些基本概念会简单介绍一下的。

本部分内容

- 第 17 章　有关事务的楔子
- 第 18 章　群雄逐鹿下的 Java 事务管理
- 第 19 章　Spring 事务王国的架构
- 第 20 章　使用 Spring 进行事务管理
- 第 21 章　Spring 事务管理之扩展篇

第17章 有关事务的楔子

本章内容

- 认识事务本身
- 初识事务家族成员

17.1 认识事务本身

要说明“什么是事务”，我觉得，先从事务存在的目的说起，会比较容易切入。

对于一个软件系统来说，需要相应的数据资源（比如，数据库、文件系统等）来保存系统状态。在对系统状态所依托的数据资源进行访问的时候，为了保证系统始终处于“正确”的状态，我们必须对这些访问操作进行一些必要的限定，以此来保证系统状态的完整性。

事务就是以可控的方式对数据资源进行访问的一组操作。为了保证事务执行前后，数据资源所承载的系统状态始终处于“正确”状态，事务本身持有4个限定属性，即原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）和持久性（Durability），这也就是常说的事务的ACID属性。

1. 原子性

原子性要求事务所包含的全部操作是一个不可分割的整体，这些操作要么全部提交成功，要么只要其中一个操作失败，就全部失败。如果把整个事务的操作比作“钢七连”，那么我们的口号就得从“不抛弃，不放弃”改成“要么不抛弃，要么就全部放弃”了。

2. 一致性

一致性要求事务所包含的操作不能违反数据资源的一致性检查，数据资源在事务执行之前处于某个数据一致性状态，那么，事务执行之后也依然需要保持数据间的一致性状态。对于一个证券系统来说，如果顾客的银行账户和证券账户资金总和为10万元的话（银行账户初始8万元，证券账户初始2万元），从银行账户的8万元转账5万元到证券账户的事务操作结束之后，银行账户会剩余3万元，证券账户为7万元，两个账户的总和依然是10万元。如果事务操作结束后，整个数据状态不是这个样子，那么就说系统处于不一致状态。使用事务的其中一个目的，就是为了避免这种不一致性状态的产生。

3. 隔离性[①]

事务的隔离性主要规定了各个事务之间相互影响的程度。隔离性概念主要面向对数据资源的并发访问（Concurrency），并兼顾影响事务的一致性。当两个事务或者更多事务同时访问同一个数据资源的时候，不同的隔离级别决定了各个事务对该数据资源访问的不同行为。

① 可称为“隔离度”，大家只需要知道我们指的都是Isolation这个概念就可以。

如果不出意外，我们可以为事务指定4种类型的隔离级别，隔离程度从弱到强分别为Read Uncommitted、Read Committed、Repeatable Read和Serializable。

- Read Uncommitted。它是最低的隔离级别。Read Uncommitted最直接的效果就是，一个事务可以读取另一个事务没有提交的更新结果。Read Uncommitted是以较低的隔离度来寻求较高的性能，其本身无法避免以下几个问题。
  - 脏读（Dirty Read）。如果一个事务对数据进行了更新，但事务还没有提交，另一个事务就可以“看到”该事务没有提交的更新结果。这样造成的问题就是，如果第一个事务回滚，那么第二个事务在此之前所“看到”的数据就是一笔脏数据。
  - 不可重复读取（Non-Repeatable Read）。不可重复读取是指同一个事务在整个事务过程中对同一笔数据进行读取，每次读取结果都不同。如果事务1在事务2的更新操作之前读取一次数据，在事务2的更新操作之后再读取同一笔数据一次，两次结果是不同的。所以，Read Uncommitted也无法避免不可重复读取的问题。
  - 幻读（Phantom Read）[①]。幻读是指同样一个查询在整个事务过程中多次执行后，查询所得的结果集是不一样的。幻读针对的是多笔记录。在Read Uncommitted隔离级别下，不管事务2的插入操作是否提交，事务1在插入操作之前和之后执行相同的查询，取得的结果集是不同的，所以，Read Uncommitted同样无法避免幻读的问题。
- Read Committed。通常，Read Committed是大部分数据库采用的默认隔离级别，它比Read Uncommitted隔离级别拥有更高级别的限定。在该隔离级别下，一个事务的更新操作结果只有在该事务提交之后，另一个事务才可能读取到同一笔数据更新后的结果。所以，Read Committed可以避免Read Uncommitted隔离级别下存在的脏读问题，但是无法避免不可重复读取和幻读的问题。
- Repeatable Read。Repeatable Read隔离级别可以保证在整个事务的过程中，对同一笔数据的读取结果是相同的，不管其他事务是否同时在对同一笔数据进行更新，也不管其他事务对同一笔数据的更新提交与否。Repeatable Read隔离级别避免了脏读和不可重复读取的问题，但是无法避免幻读。
- Serializable。它是最为严格的隔离级别。所有的事务操作都必须依次顺序执行，可以避免其他隔离级别遇到的所有问题，是最为安全的隔离级别。但同时也是性能最差的隔离级别，因为在该隔离级别下，所有的事务都需要依次顺序执行，所以，并发度下降，吞吐量上不去，性能自然就下来了。因为该隔离级别极大影响系统性能，所以，很少场景会使用它。通常情况下，我们会使用其他隔离级别加上相应的并发锁的机制来控制对数据的访问，这样既保证了系统性能不会损失太大，也能够在一定程度上保证数据的一致性。

对于数据库来说，通常都有一个默认的隔离级别，大多数情况下都是Read Committed，只有Hsqldb使用Read Uncommited作为默认隔离级别。而且，并非所有的数据库都支持这4种隔离级别，比如Oracle只支持Read Committed和Serializable。如果指定的隔离级别当前数据库不支持，那么数据库会采用默认的隔离级别代替我们指定的隔离级别。EJB、Spring、JDBC等数据访问方式，都允许我们为事务指定以上提到的4种隔离级别，但最终事务是否以指定的隔离级别执行，则由底层的数据资源来决定。

不同的隔离级别设置会对系统的并发性以及数据一致性造成不同的影响。总地来说，隔离级别与系统并发性成反比，与数据一致性成正比。也就是说，事务隔离度越高，系统并发性越差，进而造成系统性能就越差。不过，隔离度的增高，却可以更好地保证数据的一致性。隔离程度与并发性和一致

① 又称“幻影读”。

性的关系如图17-1所示。

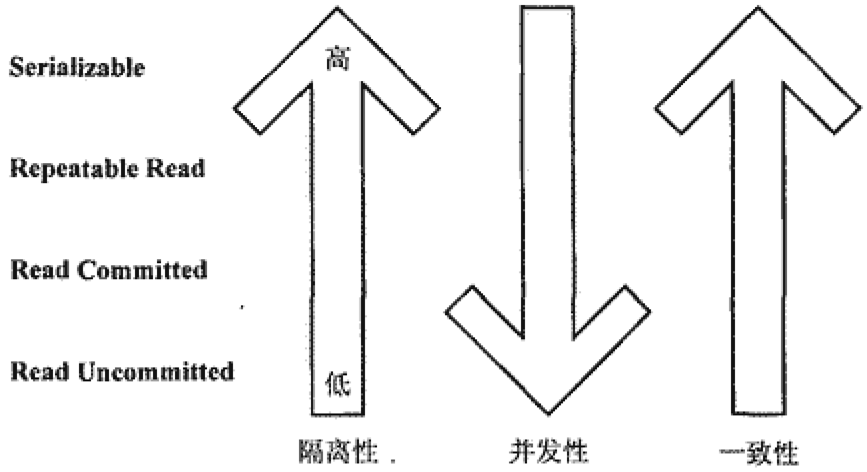

图17-1　隔离程度与并发性、一致性之间的关系

在具体的实践过程中，我们需要根据系统的具体情况来调整隔离度，保证系统性能与数据一致性之间有一个良好的平衡。总地来说，保证数据的一致性的考虑，应该优于对系统性能的考虑。

4. 持久性

事务的持久性是指，一旦整个事务操作成功提交，对数据所做的变更将被记载并不可逆转，多少有点儿“生米煮成熟饭”的意思。即使发生某些系统灾难或者什么天灾人祸之类的事情，之前事务所做的变更也可以找回并恢复。纵使海枯石烂，我（数据库等资源管理系统）对你（事务）的真心永不变！通常情况下，数据库等数据资源管理系统会通过冗余存储或者多数据网络备份等方式，来保证事务的持久性。

至此，对事务自身的认识就算告一段落了。接着，让我们进一步认识一下与事务有关的“家族成员”吧！

17.2　初识事务家族成员

在一个典型的事务处理场景中，有以下几个参与者。

- Resource Manager。简称RM，它负责存储并管理系统数据资源的状态，比如数据库服务器、JMS消息服务器等都是相应的Resource Manager。
- Transaction Processing Monitor。简称TPM或者TP Monitor，它的职责是在分布式事务场景中协调包含多个RM的事务处理。TP Monitor通常对应特定的软件中间件（Middleware）。随着软件开发技术的进步，TP Monitor的实现也由原来基于过程式的设计与实现，转向面向对象的更趋模块化的设计和实现。通常，J2EE规范[①]中的应用服务器（Application Server）担当的就是TP Monitor的角色。
- Transaction Manager。简称为TM，它可以认为是TP Monitor中的核心模块，直接负责多RM之间事务处理的协调工作，并且提供事务界定（Transaction Demarcation）、事务上下文传播（Transaction Context Propagation）等功能接口。
- Application。以独立形式存在的或者运行于容器中的应用程序，可以认为是事务边界的触发点。

实际上，并非每个事务的场景中都会出现以上提到的所有参与者。如果根据整个事务中涉及的RM

① 更确切地说是JTA（Java Transaction API）以及JTS规范（Java Transaction Service Specification）。

的多寡来区分事务类型的话，可以将事务分为两类，即全局事务（Global Transaction）和局部事务（Local Transaction）。在这两类事务中，具体的事务参与者是不同的。

全局事务。如果整个事务处理过程中有多个RM参与，那么就需要引入TP Monitor来协调多个RM之间的事务处理。TP Monitor将采用两阶段提交（Two-Phase Commit）协议来保证整个事务的ACID属性。这种场景下的事务，就称为全局事务或者分布式事务（Distributed Transaction）。全局事务中各个参与者之间的关系如图17-2所示。

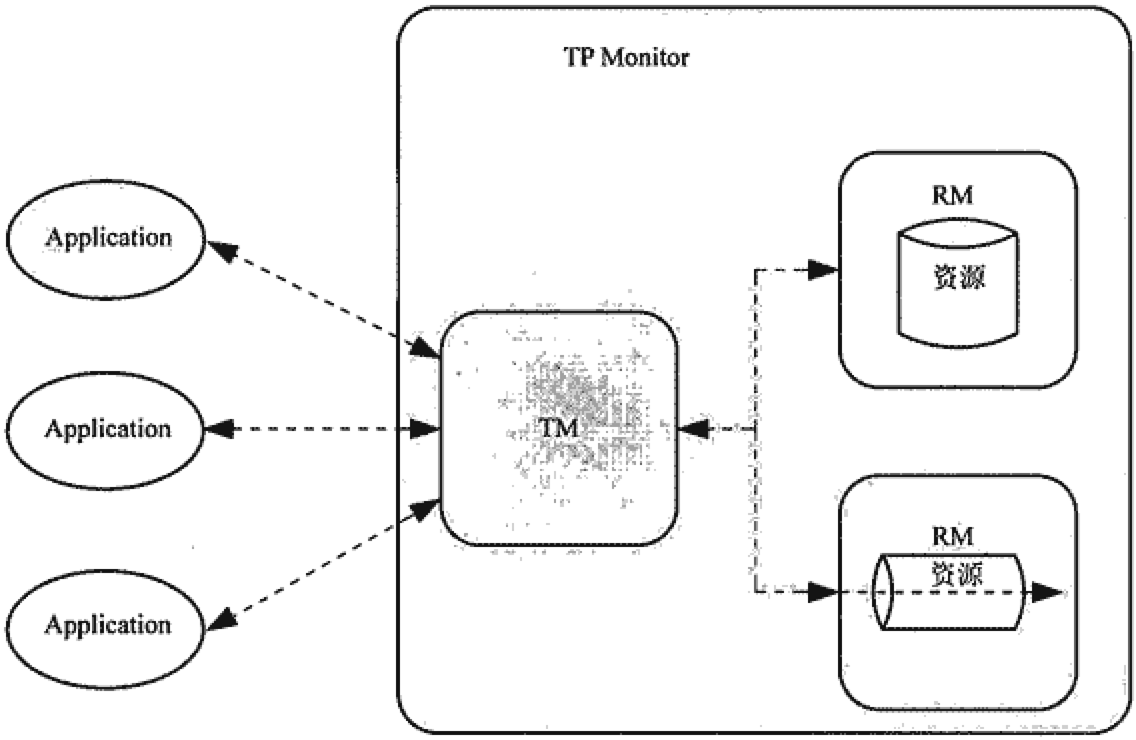

图17-2 全局事务示意图

所有应用程序提交的事务请求，需要通过TP Monitor的调配之后，直接由TM统一协调，TM将使用两阶段提交协议来协调多RM之间的事务处理。针对两阶段提交的描述，最经典的比喻就是西方婚礼的过程，婚礼的牧师或者主持是TM，他会首先分别询问两位新人（两个RM），是否愿意娶对方为妻或者嫁给对方。如果双方的反馈都是“I do”，那么牧师将宣布二者结为夫妻（即整个事务提交成功）。如果双方任何一方有疑议，那么不好意思，婚礼无法继续进行，整个事务提交失败，双方都要回滚（Rollback）到之前的单身状态。

局部事务。如果当前事务只有一个RM参与其中，我们就可以称当前事务为局部事务。比如，在当前事务中只对一个数据库进行更新，或者只向一个消息队列中发送消息的情况，都属于局部事务。局部事务的一般场景如图17-3所示。

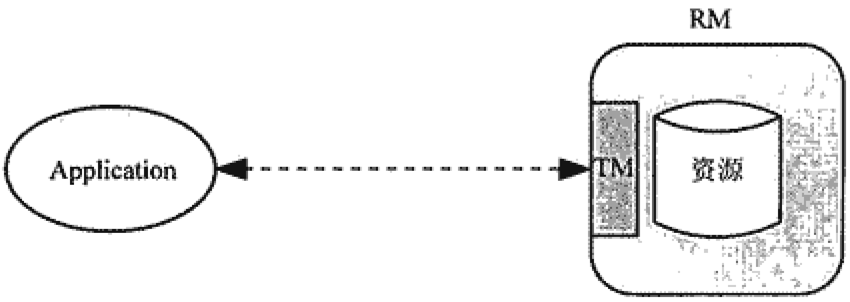

图17-3 局部事务示意图

因为局部事务只包含一个RM，所以没有必要引入相应的TP Monitor来帮助协调管理多个RM之间的事务处理，应用程序可以直接与RM打交道。通常情况下，相应的RM都有内置的事务支持，所以，在局部事务中，我们更倾向于直接使用RM的内置事务支持。这样不仅可以极大地减少事务处理的复杂度，也避免了引入TP Monitor来协调多个RM之间事务处理的性能负担。

> 小心　局部事务与全局事务的主要区分在于事务中涉及多少RM，而不是系统中实际有多少RM，这是需要我们注意的地方。即使系统中存在多个数据库（即RM），只要当前事务只更新一个数据库的数据，那么当前事务就应该算作局部事务，而不是全局事务（虽然这种情况下，我们也可以启用全局事务）。
>
> 实际上，针对单一事务资源的事务管理，我们可以在局部事务中直接使用RM内置的事务支持，也可以引入TP Monitor在分布式事务场景中进行事务管理。通常情况下，各TP Monitor在实现的时候会检测参与事务的RM数目，如果只有单一的RM参与，TP Monitor会做一定的优化，避免采用两阶段提交协议的负担。但即使如此，针对只有单一事务资源参与的事务，直接采用局部事务中RM内置的事务支持，无论是从复杂度，还是从效率上来看，都要更胜一筹。

到此为止，我们所阐述的都是概念层面的东西，要真正的在系统开发中使用事务，需要相应的产品和API支持。为了让事务“走进现实”，不同的组织或者不同的技术平台会有各自不同的API设计和实现，但既然Spring（不是Spring.Net）归属于Java一族，其他的平台和组织的解决方案暂且放于一边，我们专门来看Java平台的事务解决方案如何？

17.3 小结

在开始对Spring的事务管理层的深层进行探索之前，本章先对事务的一些基本概念和背景进行了一个总体上的回顾，这些只是更高抽象层次上的内容。下面我们将更进一步，详细看一下Java平台上对事务处理都提供了哪些支持。

第 18 章 群雄逐鹿下的Java事务管理

本章内容

- Java平台的局部事务支持
- Java平台的分布式事务支持
- 继续前行之前的反思

对于应用程序的开发人员来说，更多的时候，我们只是通过相应产品提供的API接口来访问事务资源，并考虑如何在应用的业务逻辑中界定事务边界，而对于各提供商如何在产品中实现事务支持，通常不是我们需要关心的问题。本章内容将更多地讲述各事务处理场景下，我们可以通过哪些产品提供的事务处理接口或者标准的事务处理接口来进行事务控制。当然，期间我们也可能提及相应场景下比较受欢迎的几款事务处理的产品实现。

下面将按照从局部事务场景到全局事务场景的顺序，介绍在各场景中，Java平台都为我们准备了哪些可用的事务处理API。

18.1 Java 平台的局部事务支持

在Java的局部事务场景中，系统里事务管理的具体处理方式，会随着所使用的数据访问技术的不同而各异。我们不是使用专用的事务API来管理事务，而是通过当前使用的数据访问技术所提供的基于connection[①]的API来管理事务。

数据库资源的局部事务管理。要在对数据库的访问过程中进行事务管理，每种数据访问技术都提供了特定于它自身的事务管理API，比如JDBC是Java平台访问关系数据库最基础的API。如果直接使用JDBC进行数据访问的话，我们可以将数据库连接（`java.sql.Connection`）的自动提交（AutoCommit）功能设置为`false`，改为手动提交来控制整个事务的提交或者回滚，如代码清单18-1所示。

代码清单18-1　基于JDBC API的事务管理代码示例

```
Connection connection = null;
boolean rollback = false;
try
{
    connection = dataSource.getConnection();
    connection.setAutoCommit(false);
    // 使用JDBC进行数据访问
```

① 这里的connection不是特指`java.sql.Connection`类型，而是泛指应用程序与事务资源之间的通信通道。对于JDBC来说，恰好对应`java.sql.Connection`，而对于Hibernate，那就应该是Session，诸如此类。

```
        connection.commit();
    }
    catch(SQLException e)
    {
        e.printStackTrace(); // 不要这样做
        rollback = true;
    }
    finally
    {
        if(connection != null)
        {
            if(rollback)
            {
                try {
                    connection.rollback();
                } catch (SQLException e) {
                    e.printStackTrace(); // 不要这样做
                }
            }
            else
            {
                try {
                    connection.close();
                } catch (SQLException e) {
                    e.printStackTrace(); // 不要这样做
                }
            }
        }
    }
```

而如果我们使用Hibernate进行数据访问，那就得使用Hibernate的Session进行数据访问期间的事务管理[①]，如代码清单18-2所示。

代码清单18-2　基于Hibernate API的事务管理代码示例

```
    Session session = null;
    Transaction transaction = null;
    try
    {
        session = sessionFactory.openSession();
        transaction = session.beginTransaction();
        // 使用Hibernate进行数据访问
        session.flush();
        transaction.commit();
    }
    catch(HibernateException e)
    {
        transaction.rollback();
    }
    finally
    {
        session.close();
    }
```

同样的，如果我们使用JDO、TopLink甚至JPA进行数据访问的话，这些数据访问技术也都在它们的数据访问API上，提供了相应的事务管理支持。

① 实际上，我们可以通过配置来决定让Hibernate使用基于JDBC Connection的局部事务管理，还是让Hibernate使用分布式的事务管理。

消息服务资源的局部事务管理。 在使用JMS进行消息处理的过程中，我们可以通过JMS的javax.jms.Session来控制整个处理过程的事务，如代码清单18-3所示。

代码清单18-3 JMS下的事务管理代码示例

```
boolean rollback = false;
Connection  con = null;
Session session = null;
try
{
    con = cf.createConnection();
    session = con.createSession(true, Session.AUTO_ACKNOWLEDGE);
    // 使用JMS API处理响应消息
    session.commit();
}
catch (JMSException e) {
    e.printStackTrace();          // 不要这样做
    rollback = true;
}
finally
{
    if(con != null)
    {
        if(rollback)
        {
            try {
                session.rollback();
            } catch (JMSException e1) {
                e1.printStackTrace(); // 不要这样做
            }
        }
        else
        {
            try
            {
                con.close();
            } catch (JMSException e) {
                e.printStackTrace(); // 不要这样做
            }
        }
    }
}
```

在通过javax.jms.Connection的createSession方法创建javax.jms.Session的时候，我们将该方法的第一个参数指定为true，要求创建一个事务型的javax.jms.Session实例，然后就可以根据情况提交和回滚事务了。

以上两种情况之外的时候，我们可能需要借助于JCA（Java Connector Architecture）来管理局部事务。JCA允许通过javax.resource.spi.LocalTransaction接口公开局部事务控制接口。但是，JCA只能与Java EE的应用服务器集成，所以，通过JCA访问事务资源的应用程序需要绑定到相应的Java EE服务器。

18.2 Java平台的分布式事务支持

Java平台上的分布式事务管理，主要是通过JTA（Java Transaction API）或者JCA（Java Connector Architecture）提供支持的。

18.2.1　基于 JTA 的分布式事务管理

JTA是Sun公司提出的标准化分布式事务访问的Java接口规范。不过，JTA规范定义的只是一套Java接口定义，具体的实现留给了相应的提供商去实现，各Java EE应用服务器需要提供对JTA的支持。另外，除了可以使用绑定到各Java EE应用服务器的JTA实现之外，Java平台上也存在几个独立的并且比较成熟的JTA实现产品，这包括：

- JOTM[①]
- Atomikos[②]
- JBoss Transactions[③]

使用JTA进行分布式事务管理通常有两种方式，直接使用JTA接口的编程事务管理以及基于应用服务器的声明性事务管理。

1. JTA编程事务管理

使用JTA进行分布式事务的编程式事务管理，通常使用`javax.transaction.UserTransaction`接口进行，各应用服务器都提供了针对它的JNDI查找服务。

代码清单18-4中是典型的使用`UserTransaction`进行事务管理的代码片段。

代码清单18-4　使用JTA进行分布式事务管理的代码示例

```
try
{
    UserTransaction ut = (UserTransaction)ctx.lookup("javax.transaction.UserTransaction");
    ut.begin();

    // 事务操作

    ut.commit();
} catch (NamingException e) {
    e.printStackTrace();
} catch (NotSupportedException e) {
    e.printStackTrace();
} catch (SystemException e) {
    e.printStackTrace();
} catch (SecurityException e) {
    e.printStackTrace();
} catch (IllegalStateException e) {
    e.printStackTrace();
} catch (RollbackException e) {
    e.printStackTrace();
} catch (HeuristicMixedException e) {
    e.printStackTrace();
} catch (HeuristicRollbackException e) {
    e.printStackTrace();
}
```

如果是在EJB中使用`UserTransaction`的话，我们可以直接通过`javax.ejb.EJBContext`获得`UserTransaction`的引用，如下所示：

```
UserTransaction ut = ctx.getUserTransaction();
...
```

---

① http://jotm.objectweb.org/index.html。

② http://www.atomikos.com/home.html。

③ http://labs.jboss.com/jbosstm/。

在EJB中直接编程使用UserTransaction，主要是在BMT（Bean Managed Transaction）的情况下。不过，在EJB中，有一种比直接使用编程方式更吸引人的事务管理方式，那就是在CMT（Container Managed Transacton）情况下，EJB容器提供的声明性事务管理。

2. JTA声明性事务管理

如果使用EJB进行声明性的分布式事务管理的话（限于CMT的情况），JTA的使用则只限于EJB容器内部，对于应用程序来说则完全就是透明的。现在唯一需要做的工作实际上就是在相应的部署描述符中指定相应的事务属性即可，如代码清单18-5所示。

代码清单18-5 CMT下的EJB的声明性事务管理配置代码示例

```
<enterprise-beans>
    <session>
        <display-name>Your Enterprise Java Bean</display-name>
        <ejb-name>YourBean</ejb-name>
        <home>...</home>
        <remote>...</remote>
        <ejb-class>...YourBean</ejb-class>
        <session-type>Stateless</session-type>
        <transaction-type>Container</transaction-type>
    </session>
</enterprise-beans>
<assembly-descriptor>
    <container-transaction>
        <method>
            <ejb-name>YourBean</ejb-name>
            <method-name>*</method-name>
        </method>
        <trans-attribute>Required</trans-attribute>
    </container-transaction>
</assembly-descriptor>
```

我们通过<transaction-type>指定了让EJB容器来管理事务，并且<trans-attribute>规定所有方法执行都需要相应事务支持。现在，应用程序内部再也不用充斥着各种事务管理的相关代码了。

18.2.2 基于 JCA 的分布式事务管理

JCA规范主要面向EIS（Enterprise Information System）的集成，通过为遗留的EIS系统和Java EE应用服务器指定统一的通信标准，二者就可以实现各种服务上的互通。

在应用服务器通过JCA将一些EIS系统集成进来之后，我们就可以让EIS系统中的事务资源也加入到Java EE应用的全局事务中来。实际上，要在应用程序中控制这种跨越多个系统的分布式事务，我们最终还是通过JTA来进行的，JCA更多的是提供资源的集成。从这一点上来说，在Java平台上管理分布式事务，JTA是唯一的标准接口。

18.3 继续前行之前的反思

我们也看到了，Java平台提供的事务管理API足够丰富，可谓高中低档一应俱全。这非常有助于我们根据合适场景选用合适的事务管理API，但是在实际使用过程中，过多的事务管理策略的选择也会造成一些问题。

1. 局部事务的管理绑定到了具体的数据访问方式

使用JDBC进行数据访问，需要通过java.sql.Connection来控制局部事务。使用Hibernate进行

数据访问，则需要通过`org.hibernate.Session`和`org.hibernate.Transaction`来管理局部事务。使用其他的数据访问方式，自然也是要使用它们特定的数据访问API来控制局部事务。

这样导致的问题就是，事务管理代码与数据访问代码甚至业务逻辑代码相互混杂，因为局部事务场景下，我们使用的是数据访问API进行事务控制。实际使用中，如果不能通过合适的方式将事务管理的代码与数据访问代码或者业务逻辑代码进行逻辑上的隔离，将直接导致数据访问代码和业务逻辑代码的可重用性降低，甚至事务管理代码在数据访问层和业务服务层的到处散落。

当前的情况是，各种数据访问方式只提供了简单的事务API，但没有更高层次的抽象来帮助我们隔离事务与数据访问两个方面的过紧耦合。

2. 事务的异常处理

事务处理过程中出现的异常应该都是不可恢复的，所以，应该抛出unchecked exception，并且有一个统一的父类，便于客户端处理。但是现在的情况如下所述。

- 没有一个统一的事务相关异常体系。使用特定API管理事务的时候，我们需要捕捉这些 API特定的异常并进行处理。
- 许多事务管理代码在使用过程中抛出的依然还是checked exception，这将强制客户端代码来捕捉并处理它。从`UserTransaction`的使用上我们可以看出，从JNDI查找到事务的开始和结束等操作，简单的事务界定操作，却引出七八个异常需要处理。任何人在使用`UserTransaction`进行编程式事务管理的时候，也不会认为这样的API设计很好用吧？

3. 事务处理API的多样性

对于开发人员来说，所谓对事务的管理，最多也就是界定一下事务的边界，规定事务在什么地方开始，在什么地方结束。可是，要达到这个目的，我们却要在各种数据访问API或者JTA之间徘徊。

各种事务管理API的存在给了我们更多选择，但没有一个统一的方式来抽象单一的事务管理需求，反而让这种多种选择的优势变得繁杂而不易管理。

4. CMT声明式事务的局限

EJB容器提供的CMT特性是比较受欢迎的事务界定管理方式，因为业务对象中不需要混杂任何事务管理的相关代码，所有的事务管理都通过一些简单的配置交由容器来管理。CMT提供的声明式的事务管理，很好地分离了事务管理与具体的数据资源访问之间的耦合，使得开发人员能够分别专心于业务逻辑和数据访问逻辑的开发。

但CMT形式的声明式事务管理有一个令人惋惜的限制，那就是，你必须借助于EJB容器才能得到这种声明式事务的好处。如果我们的应用程序想要使用声明式的事务管理，就不得不花费一些银子来购买应用服务器厂商的授权，或者，即使是使用开源的应用服务器，但我们的应用程序在此之前并没有特别强烈的必须使用应用服务器的需求，这些情况下，引入应用服务器的做法应该都不太容易让人接受。

总之，要使用CMT的声明式事务管理，强制要求引入EJB容器的支持（当然，也得以EJB的形式提供组件实现），在某些情况下是不合适的。

鉴于这些问题，我们应该考虑对目前的状况进行改进，以便简化开发过程，提高整个过程中的开发效率。

- 我们能否对众多的基于数据访问API的局部事务操作进行合理的抽象，以便隔离事务管理与数据资源访问之间的过分耦合？
- 能否在合适的地方将各种场景下事务处理抛出的checked exception进行合适的转译，屏蔽事务处理过程中因使用不同的事务API所造成的差异性？

- 既然对于我们来说，事务管理的需求很简单，基本上就是事物界定的工作，我们能否对事务的界定操作进行统一抽象，以屏蔽各种事务管理API的差异性，使得事务管理能够以统一的编程模型来进行？
- 既然声明式的事务管理如此诱人，那么能否突破必须依赖EJB容器的限制，寻求一种能够为普通的Java对象（POJO）提供声明式事务的方式呢？

如果你也在思考这些问题，那么，恭喜你，你不用从头去探索这些问题的解决方法了，因为Spring的事务抽象框架正是我们所要寻找的解决方案。

18.4 小结

我们一起回顾了Java平台上事务管理的相关内容，分析了Java平台都为我们提供了哪些管理局部事务和全局事务的支持。我们也对平台上现有的事务管理支持进行了剖析，表达了我们想要改进的一些想法。接下来，我们将一起开始真正的Spring事务抽象层之旅，看一下Spring的事务抽象层是如何帮助我们“圆梦”的。

第19章

Spring事务王国的架构

本章内容

- 统一中原的过程
- 和平年代

Spring的事务框架将开发过程中事务管理相关的关注点进行适当的分离，并对这些关注点进行合理的抽象，最终打造了一套使用方便，却功能强大的事务管理“利器”。通过Spring的事务框架，我们可以按照统一的编程模型来进行事务编程，却不用关心所使用的数据访问技术以及具体要访问什么类型的事务资源。并且，Spring的事务框架与Spring提供的数据访问支持可以紧密结合，更是让我们在事务管理与数据访问之间游刃有余。而最主要的是，结合Spring的AOP框架，Spring的事务框架为我们带来了原来只有CMT才有的声明式事务管理的特殊待遇，却无需绑定到任何的应用服务器上。

其他溢美之词咱就先放一边，还是赶快进入正题吧！

Spring的事务框架设计理念的基本原则是：让事务管理的关注点与数据访问关注点相分离。

- 当在业务层使用事务的抽象API进行事务界定的时候，不需要关心事务将要加诸于上的事务资源是什么，对不同的事务资源的管理将由相应的框架实现类来操心。
- 当在数据访问层对可能参与事务的数据资源进行访问的时候，只需要使用相应的数据访问API进行数据访问，不需要关心当前的事务资源如何参与事务或者是否需要参与事务。这同样将由事务框架类来打理。

在以上两个关注点被清晰地分离出来之后，对于我们开发人员来说，唯一需要关心的，就是通过抽象后的事务管理API对当前事务进行界定而已，如代码清单19-1所示。

代码清单19-1　使用事务管理抽象API进行事务界定的代码示例

```
public class FooService
{
    private PlatformTransactionManager transactionManager;

    public void serviceMethod()
    {
        TransactionDefinition definition = ...;
        TransactionStatus txStatus = getTransactionManager().getTransaction(definition);
        try
        {
            // dao1.doDataAccess();
            // dao2.doDataAccess();
            // ...
        }
        catch(DataAccessException e)
        {
```

```
            getTransactionManager().rollback(txStatus);
            throw e;
        }
        catch(OtherNecessaryException e)
        {
            getTransactionManager().rollback(txStatus);
            throw e;
        }
        getTransactionManager().commit(txStatus);
    }

    public PlatformTransactionManager getTransactionManager() {
        return transactionManager;
    }

    public void setTransactionManager(PlatformTransactionManager transactionManager) {
        this.transactionManager = transactionManager;
    }
}
```

不管数据访问方式如何变换，事务管理实现也可以岿然不动。只要有了这样一个统一的事务管理编程模型，剩下的声明式事务管理也就很自然地成为锦上添花之作了！从此之后，事务管理就是事务管理，数据访问只关心数据访问，再也不用因为它们之间的纠缠而烦恼。

19.1 统一中原的过程

org.springframework.transaction.PlatformTransactionManager是Spring事务抽象架构的核心接口，它的主要作用是为应用程序提供事务界定的统一方式。既然事务界定的需要很简单，那么PlatformTransactionManager的定义看起来也不会过于复杂，如下所示：

```
public interface PlatformTransactionManager {
    TransactionStatus getTransaction(TransactionDefinition definition) throws ➥
    Transaction Exception;
    void commit(TransactionStatus status) throws TransactionException;
    void rollback(TransactionStatus status) throws TransactionException;
}
```

PlatformTransactionManager是整个事务抽象策略的顶层接口，它就好像我们的战略蓝图，而战略的具体实施则将由相应的PlatformTransactionManager实现类来执行。

Spring的事务框架针对不同的数据访问方式以及全局事务场景，提供了相应的Platform-TransactionManager实现类。在每个实现类的职责完成之后，Spring事务框架的“统一大业”就完成了。在深入了解各个PlatformTransactionManager实现类的奥秘之前，不妨先考虑一下，如果让我们来实现一个PlatformTransactionManager，要如何去做？

不妨先以针对JDBC数据访问方式的局部事务管理为例。对于层次划分清晰的应用来说，我们通常都是将事务管理放在Service层，而将数据访问逻辑放在DAO层。这样做的目的是，可以不用因为将事务管理代码放在DAO层，而降低数据访问逻辑的重用性，也可以在Service层根据相应逻辑，来决定提交或者回滚事务。一般的Service对象可能需要在同一个业务方法中调用多个数据访问对象的方法，类似于图19-1这样的情况。

因为JDBC的局部事务控制是由同一个java.sql.Connection来完成的，所以要保证两个DAO的数据访问方法处于一个事务中，我们就得保证它们使用的是同一个java.sql.Connection。要做到这一点，通常会采用称为connection-passing的方式，即为同一个事务中的各个dao的数据访问方法传递当

前事务对应的同一个java.sql.Connection。这样，我们的业务方法以及数据访问方法都得做一定的修改，如图19-2所示。

```
public void serviceMethod()
{
        Object txObject=transactionManager.beginTransaction();
        ...
        dao1.doDataAccess();
        dao2.doDataAccess();
        ...
        transactionManager.commitTransaction(txObject);
}
```

图19-1 普通情况下的事务管理代码

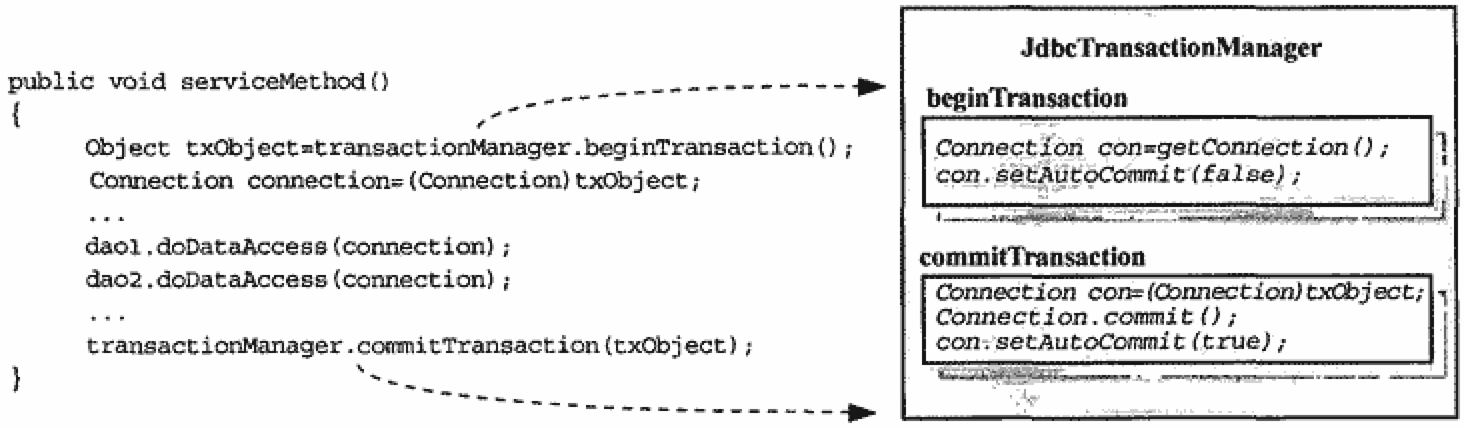

图19-2 connection-passing方式的事务管理代码

我们只要把java.sql.Connection的获取并设置Autocommit状态的代码，以及使用java.sql.Connection提交事务的代码，重构到原来的开启事务以及提交事务的方法中，针对JDBC的局部事务管理的整合看起来离成功也就是咫尺之遥了。不过，这看起来的咫尺之遥，实际上却依然遥远。

使用这种方式，最致命的一个问题就是，不但事务管理代码无法摆脱java.sql.Connection的纠缠，而且数据访问对象的定义要绑定到具体的数据访问技术上来。现在是使用JDBC进行数据访问，我们要在数据访问方法中声明对java.sql.Connection的依赖，那要是使用Hibernate的话，是不是又要声明对Session的依赖呢？显然，这样的做法是不可行的。不过好消息是，传递Connection的理念是对的，只不过，在具体实施过程中，我们所采用的方法有些不对头。

要传递java.sql.Connection，我们可以将整个事务对应的java.sql.Connection实例放到统一的一个地方去，无论是谁，要使用该资源，都从这一个地方来获取，这样就解除了事务管理代码和数据访问代码之间通过java.sql.Connection的“直接”耦合。具体点儿说就是，我们在事务开始之前取得一个java.sql.Connection，然后将这个Connection绑定到当前的调用线程。之后，数据访问对象在使用Connection进行数据访问的时候，就可以从当前线程上获得这个事务开始的时候绑定的Connection实例。当所有的数据访问对象全部使用这个绑定到当前线程的Connection完成了数据访问工作时，我们就使用这个Connection实例提交或者回滚事务，然后解除它到当前线程的绑定。整个过程如图19-3所示。

这时的java.sql.Connection就像那大河上的一条船，从启航（事务开始）到航程结束（事务完

成）这整个期间，大河沿岸都可以与该船打交道。而至于说是发“木船”还是发“铁轮”，那要根据情况来决定了：发JDBC的船，那就是Connection；发Hibernate的船，那就是Session……

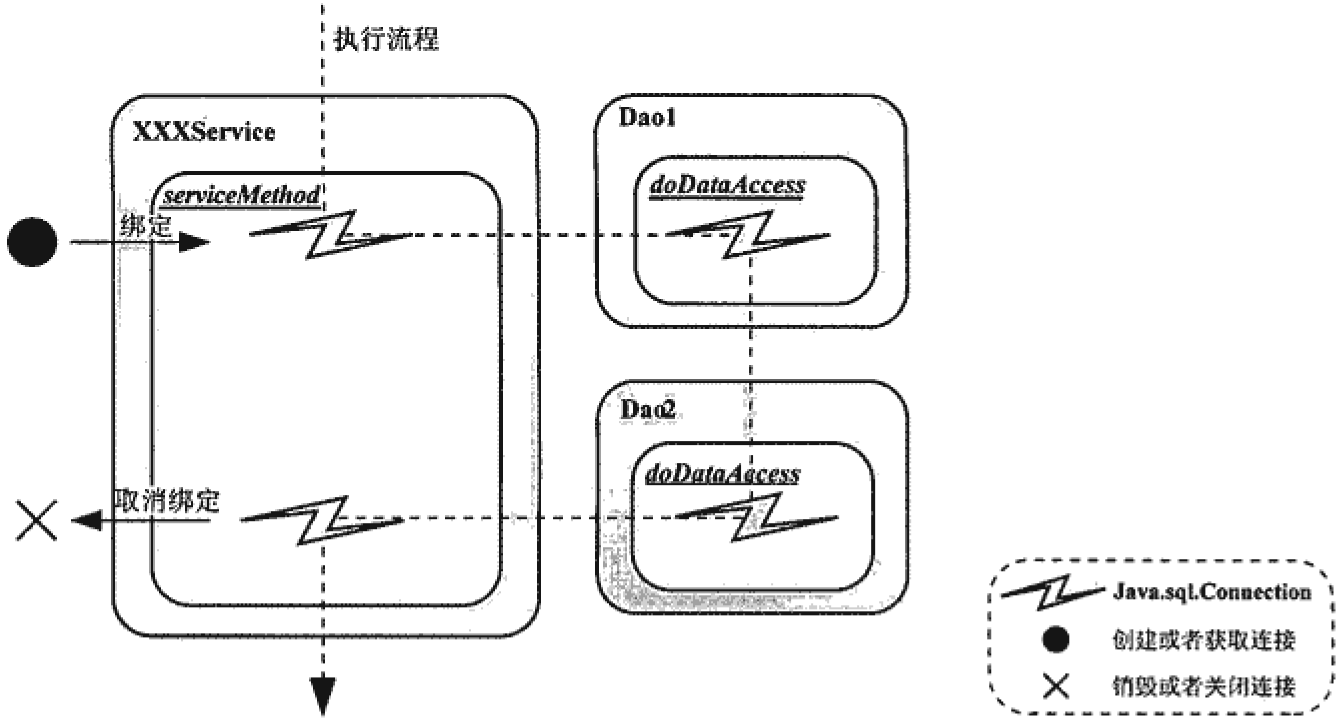

图19-3 java.sql.Connection绑定到线程示意图

假设TransactionResourceManager就是存放java.sql.Connection（或者其他事务资源）的地方，那么，它看起来将如代码清单19-2所示（过多的逻辑检验代码略去）。

代码清单19-2 TransactionResourceManager原型定义代码示例

```
public class TransactionResourceManager
{
    private static ThreadLocal resources = new ThreadLocal();

    public static Object getResource()
    {
        return resources.get();
    }

    public static void bindResource(Object resource)
    {
        resources.set(resource);
    }

    public static Object unbindResource()
    {
        Object res = getResource();
        resources.set(null);
        return res;
    }
}
```

对于我们要实现的针对JDBC的PlatformTransactionManager，只需要在事务开始的时候，通过我们的TransactionResourceManager将java.sql.Connection绑定到线程，然后在事务结束的时候解除绑定即可（原型代码参照代码清单19-3）。

代码清单19-3　面向JDBC的PlatformTransactionManager原型实现代码示例

```
public class JdbcTransactionManager implements PlatformTransactionManager
    {
    private DataSource dataSource;

    public JdbcTransactionManager(DataSource dataSource)
    {
        this.dataSource = dataSource;
    }

    public TransactionStatus getTransaction(TransactionDefinition definition) ➥
    throws TransactionException {
            Connection connection;
        try
        {
            connection = dataSource.getConnection();
            TransactionResourceManager.bindResource(connection);
            return new ➥
            DefaultTransactionStatus(connection,true,true,false,true,null);
        }
        catch (SQLException e)
        {
            throw new ➥
            CannotCreateTransactionException("can't get connection for tx",e);
        }
    }

    public void rollback(TransactionStatus txStatus) throws TransactionException {
        Connection connection = ➥
        (Connection)TransactionResourceManager.unbindResource();
        try
        {
            connection.rollback();
        }
        catch (SQLException e)
        {
            throw new ➥
            UnexpectedRollbackException("rollback failed with SQLException",e);
        }
        finally
        {
            try {
                connection.close();
            } catch (SQLException e) {
                // 记录异常信息，但通常不会有进一步有效的处理
            }
        }
    }
    public void commit(TransactionStatus txStatus) throws TransactionException {
        Connection connection = ➥
        (Connection)TransactionResourceManager.unbindResource();
        try
        {
            connection.commit();
        }
        catch (SQLException e)
        {
            throw new ➥
            TransactionSystemException("commit failed with SQLException",e);
        }
```

```
        finally
        {
            try {
                connection.close();
            } catch (SQLException e) {
                // 记录异常信息，但通常不会有进一步有效的处理
            }
        }
    }
}
```

因为Connection在事务开始和结束期间都可以通过我们的TransactionResourceManager获得，所以所有的DAO层数据访问对象在使用JDBC进行数据访问的时候，就可以直接从Transaction-ResourceManager获得数据库连接并进行数据访问。这样就可以保证在整个事务期间，所有的数据访问对象对应的是同一个Connection，如下所示：

```
public class FooJdbcDao implements IDao{
    public void doDataAccess(){
        Connection con = (Connection)TransactionResourceManager.getResource();
        // ...
    }
}
```

至此，我们完成了PlatformTransactionManager的具体实现类，并解除了它与相应数据访问对象之间通过java.sql.Connection的直接耦合。进行事务控制的时候，我们只需要为Service对象提供相应的PlatformTransactionManager实现类，Service对象中的事务管理功能就算大功告成了，而不需要关心到底对应的是什么样的事务资源，甚至什么样的数据访问方式。

当然，为了便于你理解Spring抽象层的实现原理，以上的代码实例都是简化后的模型，所以，不要试图将它们应用于生产环境。原型代码永远都是原型代码，要做的事情还有许多，如下所述。

(1) 如何保证PlatformTransactionManager的相应方法以正确的顺序被调用？如果哪个方法没有被正确调用，也会造成资源泄漏以及事务管理代码混乱的问题。在稍后介绍使用Spring进行编程事务管理的时候，你将看到Spring是如何解决这个问题的。

(2) 数据访问对象的接口定义不会因为最初的connection-passing方式而改变契约了，但是，现在却要强制每个数据访问对象使用TransactionResourceManager来获取数据资源接口。另外，如果当前数据访问对象对应的数据方法不想参与跨越多个数据操作的事务的话，甚至于不想（或不能）使用类似的事务管理支持，是否就意味着无法获得connection进行数据访问了呢？

不知道你是否还记得我们在介绍Spring的数据访问相关内容的时候，曾经提到的org.springframework.jdbc.datasource.DataSourceUtils工具类。当时我们只是强调了DataSourceUtils提供的异常转译能力。实际上，DataSourceUtils最主要的工作是对connection的管理，DataSourceUtils会从类似TransactionResourceManager的类（Spring中对应org.springframework.transaction.support.TransactionSynchronizationManager）那里获取Connection资源。如果当前线程之前没有绑定任何connection，那么它就通过数据访问对象的DataSource引用获取新的connection，否则就使用绑定的那个connection。这就是为什么要强调，当我们要使用Spring提供的事务支持的时候，必须通过DataSourceUtils来获取连接。因为它提供了Spring事务管理框架在数据访问层需要提供的基础设施中不可或缺的一部分，而JdbcTemplate等类内部已经使用DataSourceUtils来管理连接了，所以，我们不用操心这些细节。从这里，我们也应可以看出，Spring的事务管理与它的数据访问框架是紧密结合的。

> **注意**　对应Hibernate的SessionFactoryUtils，对应JDO的PersistenceManagerFactoryUtils以及对应其他数据访问技术的Utils类，它们的作用与DataSourceUtils是相似的。除了提供异常转译功能，它们更多地被用于数据访问资源的管理工作，以配合对应的PlatformTransactionManager实现类进行事务管理。

实际上，Spring在实现针对各种数据访问技术的PlatformTransactionManager的时候，要考虑很多东西，不像原型以及提出的几个问题所展示的那么简单。不过，各个实现类的基本思路与原型所展示的是基本吻合的。当我们了解了针对JDBC的PlatformTransactionManager是如何实现的时候，其他的实现类基本上就是平推了。

19.2　和平年代

Spring的事务抽象包括3个主要接口，即PlatformTransactionManager、TransactionDefinition以及TransactionStatus，它们之间的关系如图19-4所示。

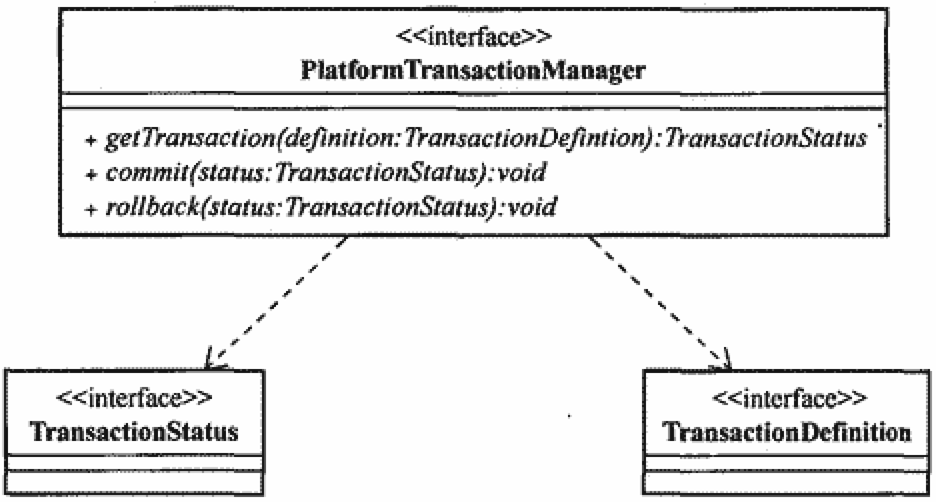

图19-4　Spring事务抽象接口关系图

这3个接口以org.springframework.transaction.PlatformTransactionManager为中心，互为犄角，多少有点儿“晋西北铁三角”的味道。org.springframework.transaction.PlatformTransactionManager负责界定事务边界。org.springframework.transaction.TransactionDefinition负责定义事务相关属性，包括隔离级别、传播行为等。org.springframework.transaction.PlatformTransactionManager将参照org.springframework.transaction.TransactionDefinition的属性定义来开启相关事务。事务开启之后到事务结束期间的事务状态由org.springframework.transaction.TransactionStatus负责，我们也可以通过org.springframework.transaction.TransactionStatus对事务进行有限的控制。

19.2.1　TransactionDefinition

1. TransactionDefinition简介

org.springframework.transaction.TransactionDefinition主要定义了有哪些事务属性可以指定，这包括：

- 事务的隔离（Isolation）级别
- 事务的传播行为（Propagation Behavior）
- 事务的超时时间（Timeout）
- 是否为只读（ReadOnly）事务

TransactionDefinition内定义了如下5个常量用于标志可供选择的隔离级别。

- ISOLATION_DEFAULT。如果指定隔离级别为ISOLATION_DEFAULT，则表示使用数据库默认的隔离级别，通常情况下是Read Committed。

- ISOLATION_READ_UNCOMMITTED。对应Read Uncommitted隔离级别，无法避免脏读，不可重复读和幻读。
- ISOLATION_READ_COMMITTED。对应Read Committed隔离级别，可以避免脏读，但无法避免不可重复读和幻读。
- ISOLATION_REPEATABLE_READ。对应Repeatable read隔离级别，可以避免脏读和不可重复读，但不能避免幻读。
- ISOLATION_SERIALIZABLE。对应Serializable隔离级别，可以避免所有的脏读，不可重复读以及幻读，但并发性效率最低。

事务的传播行为（Propagation Behavior）我们之前没有提到，如果你之前接触过EJB的CMT的话，对它应该也不会陌生。事务的传播行为表示整个事务处理过程所跨越的业务对象，将以什么样的行为参与事务（我们将在声明式事务中更多地依赖于该属性）。比如，当有FoobarService调用用FooService和BarService两个方法的时候，FooService的业务方法和BarService的业务方法可以指定它们各自的事务传播行为（如图19-5所示）。

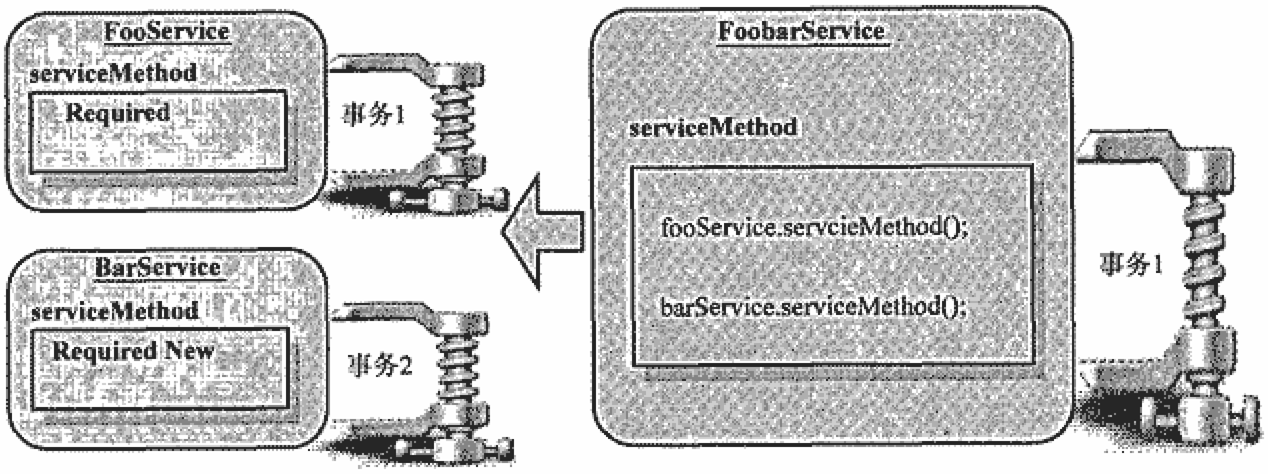

图19-5 业务方法的传播行为

FooService的业务方法的传播行为被我们指定为Required，表示如果当前存在事务的话，则加入当前事务。因为FoobarService在调用FooService的业务方法的时候已经启动了一个事务，所以，FooSerivce的业务方法会直接加入FoobarService启动的事务1中。BarService的业务方法的传播行为被指定为Required New，表示无论当前是否存在事务，都需要为其重新启动一个事务，所以，它使用的是自己启动的事务2。

针对事务的传播行为，TransactionDefinition提供了以下几种选择，除了PROPAGATION_NESTED是Spring特有的外，其他传播行为的语义与CMT基本相同。

- PROPAGATION_REQUIRED。如果当前存在一个事务，则加入当前事务。如果不存在任何事务，则创建一个新的事务。总之，要至少保证在一个事务中运行。PROPAGATION_REQUIRED通常作为默认的事务传播行为。
- PROPAGATION_SUPPORTS。如果当前存在一个事务，则加入当前事务。如果当前不存在事务，则直接执行。对于一些查询方法来说，PROPAGATION_SUPPORTS通常是比较合适的传播行为选择。如果当前方法直接执行，那么不需要事务的支持。如果当前方法被其他方法调用，而其他方法启动了一个事务，使用PROPAGATION_SUPPORTS可以保证当前方法能够加入当前事务，并洞察当前事务对数据资源所做的更新。比如，A.service()会首先更新数据库，然后调用

B.service()进行查询，那么，如果B.service()是PROPAGATION_SUPPORTS的传播行为，就可以读取A.service()之前所做的最新更新结果（如图19-6所示）。而如果使用稍后所提到的PROPAGATION_NOT_SUPPORTED，则B.service()将无法读取最新的更新结果，因为A.service()的事务在这时还没有提交（除非隔离级别是Read Uncommitted）。

- PROPAGATION_MANDATORY。PROPAGATION_MANDATORY强制要求当前存在一个事务，如果不存在，则抛出异常。如果某个方法需要事务支持，但自身又不管理事务提交或者回滚，那么比较适合使用PROPAGATION_MANDATORY。可以参照*Java Transaction Design Strategies*一书中对REQUIRED和MANDATORY两种传播行为的比较，来更深入地了解PROPAGATION_ MANDATORY可能的应用场景。

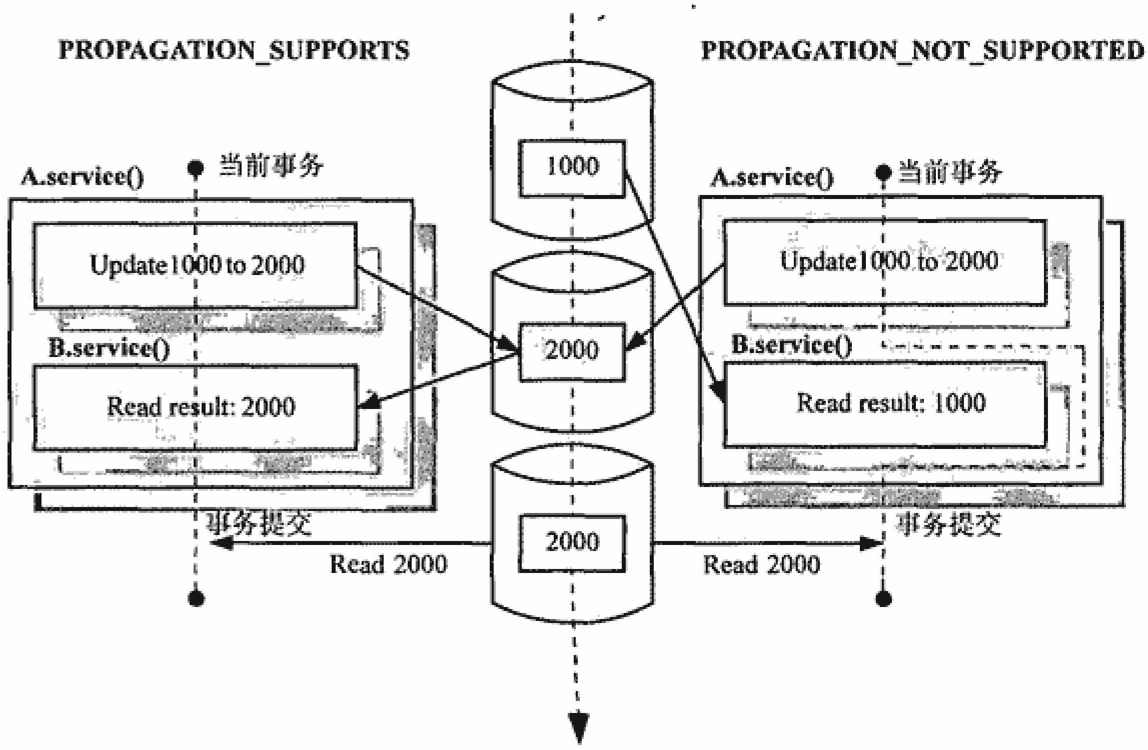

图19-6　PROPAGATION_SUPPORTS可能场景

- PROPAGATION_REQUIRES_NEW。不管当前是否存在事务，都会创建新的事务。如果当前存在事务，会将当前的事务挂起（Suspend）。如果某个业务对象所做的事情不想影响到外层事务，PROPAGATION_REQUIRES_NEW应该是合适的选择。比如，假设当前的业务方法需要向数据库中更新某些日志信息，但即使这些日志信息更新失败，我们也不想因为该业务方法的事务回滚，而影响到外层事务的成功提交。因为这种情况下，当前业务方法的事务成功与否对外层事务来说是无关紧要的。
- PROPAGATION_NOT_SUPPORTED。不支持当前事务，而是在没有事务的情况下执行。如果当前存在事务的话，当前事务原则上将被挂起（Suspend），但这要看对应的PlatformTransaction-Manager实现类是否支持事务的挂起。更多情况请参照TransactionDefinition的Javadoc文档。PROPAGATION_NOT_SUPPORTED与PROPAGATION_SUPPORTS之间的区别，可以参照PROPAGATION_SUPPORTS部分的实例内容。
- PROPAGATION_NEVER。永远不需要当前存在事务，如果存在当前事务，则抛出异常。
- PROPAGATION_NESTED。如果存在当前事务，则在当前事务的一个嵌套事务中执行，否则与PRO-

PAGATION_REQUIRED的行为类似，即创建新的事务，在新创建的事务中执行。PROPAGATION_NESTED粗看起来好像与PROPAGATION_REQUIRES_NEW的行为类似，实际上二者是有差别的。PROPAGATION_REQUIRES_NEW创建的新事务与外层事务属于同一个“档次”，即二者的地位是相同的。当新创建的事务运行的时候，外层事务将被暂时挂起。而PROPAGATION_NESTED创建的嵌套事务则不然，它是寄生于当前外层事务的，它的地位比当前外层事务的地位要小一号。当内部嵌套事务运行的时候，外层事务也是处于active状态，图19-7演示了PROPAGATION_REQUIRES_NEW和PROPAGATION_NESTED中涉及的多个事务相互之间的地位。

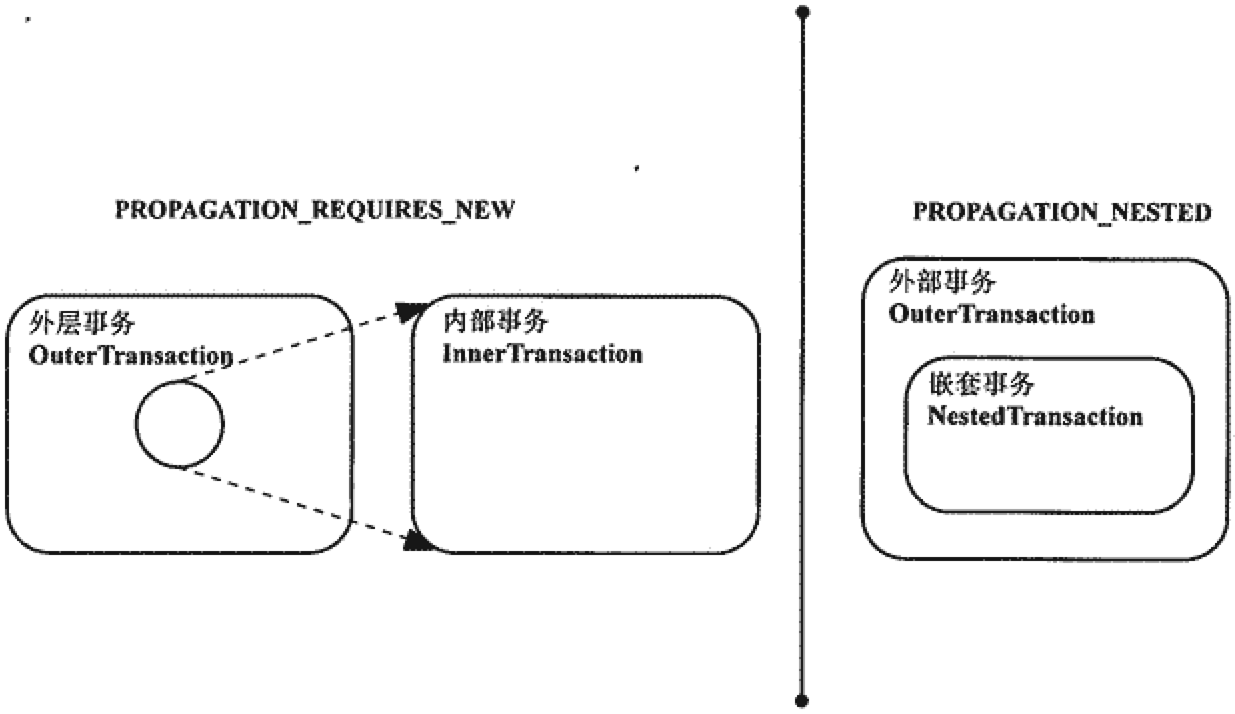

图19-7 · PROPAGATION_REQUIRES_NEW与PROPAGATION_NESTED创建的事务的区别

也就是说，虽然PROPAGATION_REQUIRES_NEW新创建的事务是在当前外层事务内执行，但新创建的事务是独立于当前外层事务而存在的，二者拥有各自独立的状态而互不干扰。而PROPAGATION_NESTED创建的事务属于当前外层事务的内部子事务（Sub-transaction），内部子事务的处理内容属于当前外层事务的一部分，不能独立于外层事务而存在，并且与外层事务共有事务状态。我想这也就是为什么称其为内部嵌套事务的原因。

PROPAGATION_NESTED可能的应用场景在于，你可以将一个大的事务划分为多个小的事务来处理，并且外层事务可以根据各个内部嵌套事务的执行结果，来选择不同的执行流程。比如，某个业务对象的业务方法`A.service()`，可能调用其他业务方法`B.service()`向数据库中插入一批业务数据，但当插入数据的业务方法出现错误的时候（比如主键冲突），我们可以在当前事务中捕捉前一个方法抛出的异常，然后选择另一个更新数据的业务方法`C.service()`来执行。这时，我们就可以把`B.service()`和`C.serivce()`方法的传播行为指定为PROPAGATION_NESTED[①]。如果用伪代码演示的话，看起来如代码清单19-4所示。

① 当然，如果只是检测单条数据插入的主键冲突，然后改为更新数据的话，更多时候，我们会直接在一个数据访问方法中解决。

代码清单19-4　嵌套事务应用代码示例

```
/**
 * PROPAGATION_REQUIRED
 */
A.service()
{
    try
    {
        // PROPAGATION_NESTED
        B.service();
    }
    catch(Exception e)
    {
        // PROPAGATION_NESTED
        C.service();
    }
}
```

不过，并非所有的PlatformTransactionManager实现都支持PROPAGATION_NESTED类型的传播行为。现在只有org.springframework.jdbc.datasource.DataSourceTransactionManager在使用JDBC 3.0数据库驱动的情况下才支持（当然，数据库和相应的驱动程序也需要提供支持）。另外，某些JtaTransactionManager也可能提供支持，但是JTA规范并没有要求提供对嵌套事务的支持。

对TransactionDefinition所提供的这几个传播行为选项的使用，最好是建立在充分理解的基础上。当然，除非特殊的场景，通常情况下，PROPAGATION_REQUIRED将是我们最常用的选择。

TransactionDefinition提供了TIMEOUT_DEFAULT常量定义，用来指定事务的超时时间。TIMEOUT_DEFAULT默认值为-1，这会采用当前事务系统默认的超时时间。我们可以通过TransactionDefinition的具体实现类提供自定义的事务超时时间。

TransactionDefinition提供的最后一个重要信息就是将要创建的是否是一个只读（ReadOnly）的事务。如果需要创建一个只读的事务的话，可以通过TransactionDefinition的相关实现类进行设置。只读的事务仅仅是给相应的ResourceManager提供一种优化的提示，但最终是否提供优化，则由具体的ResourceManager来决定。对于一些查询来说，我们通常会希望它们采用只读事务。

2. TransactionDefinition相关实现

TransactionDefinition只是一个接口定义，要为PlatformTransactionManager创建事务提供信息，需要有相应的实现类提供支持。虽然TransactionDefinition的相关实现类不多，但为了便于理解，我们依然将它们划分为“两派”，如图19-8所示。

我们将TransactionDefinition的相关实现类按照编程式事务场景和声明式事务场景划分为两个分支。这只是出于每个类在相应场景中出现的频率这一因素考虑的，而不是说声明式事务场景的实现类不能在编程式事务场景中使用。

org.springframework.transaction.support.DefaultTransactionDefinition 是 TransactionDefinition接口的默认实现类，它提供了各事务属性的默认值，并且通过它的setter方法，我们可以更改这些默认设置。这些默认值包括：

- propagationBehavior = PROPAGATION_REQUIRED
- isolationLevel = ISOLATION_DEFAULT
- timeout = TIMEOUT_DEFAULT
- readOnly = false

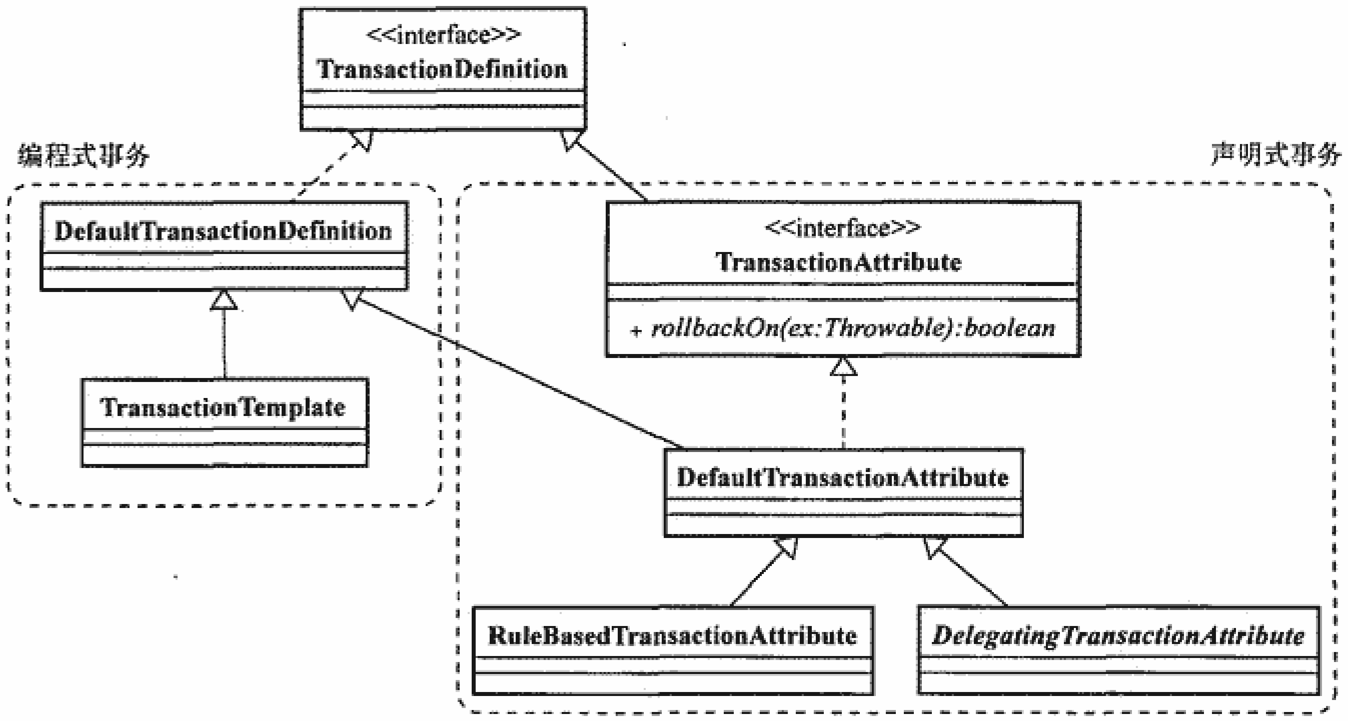

图19-8 TransactionDefinition继承层次图

org.springframework.transaction.support.TransactionTemplate是Spring提供的进行编程式事务管理的模板方法类（我们将稍后提到该类的使用），它直接继承了DefaultTransactionDefinition。所以，我们在使用TransactionTemplate的时候就可以直接通过TransactionTemplate本身提供事务控制属性。

org.springframework.transaction.interceptor.TransactionAttribute是继承自TransactionDefinition的接口定义，主要面向使用Spring AOP进行声明式事务管理的场合。它在TransactionDefinition定义的基础上添加了一个rollbackOn方法，如下所示：

```
boolean rollbackOn(Throwable ex);
```

这样，我们可以通过声明的方式指定业务方法在抛出哪些的异常的情况下可以回滚事务。

TransactionAttribute的默认实现类是DefaultTransactionAttribute，它同时继承了DefaultTransactionDefinition。在DefaultTransactionDefinition的基础上增加了rollbackOn的实现，DefaultTransactionAttribute的实现指定了，当异常类型为unchecked exception的情况下将回滚事务。

DefaultTransactionAttribute下有两个实现类，即RuleBasedTransactionAttribute和DelegatingTransactionAttribute。RuleBasedTransactionAttribute允许我们同时指定多个回滚规则，这些规则以包含org.springframework.transaction.interceptor.RollbackRuleAttribute或者org.springframework.transaction.interceptor.NoRollbackRuleAttribute的List形式提供。RuleBasedTransactionAttribute的rollbackOn将使用传入的异常类型与这些回滚规则进行匹配，然后再决定是否要回滚事务。

DelegatingTransactionAttribute是抽象类，它存在的目的就是被子类化，DelegatingTransactionAttribute会将所有方法调用委派给另一个具体的TransactionAttribute实现类，比如DefaultTransactionAttribute或者RuleBasedTransactionAttribute。不过，除非不是简单的直接委派（什么附加逻辑都不添加），否则，实现一个DelegatingTransactionAttribute是没有任

何意义的。

19.2.2 TransactionStatus

org.springframework.transaction.TransactionStatus接口定义表示整个事务处理过程中的事务状态，更多时候，我们将在编程式事务中使用该接口。

在事务处理过程中，我们可以使用TransactionStatus进行如下工作。

- 使用TransactionStatus提供的相应方法查询事务状态。
- 通过setRollbackOnly()方法标记当前事务以使其回滚。
- 如果相应的PlatformTransactionManager支持Savepoint，可以通过TransactionStatus在当前事务中创建内部嵌套事务。

在稍后将为你介绍如何使用Spring进行编程式事务管理的部分，可以更直观地了解这些。

TransactionStatus的实现层次比较简单，见图19-9。

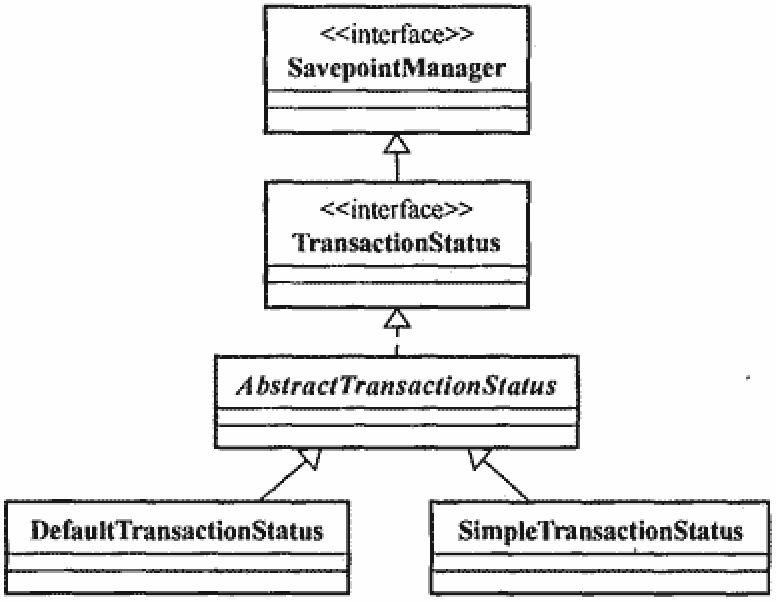

图19-9　TransactionStatus继承层次

org.springframework.transaction.SavepointManager是在JDBC 3.0的基础上，对Savepoint的支持提供的抽象。通过继承SavepointManager，TransactionStatus获得可以管理Savepoint的能力，从而支持创建内部嵌套事务。

org.springframework.transaction.support.AbstractTransactionStatus是TransactionStatus的抽象类实现，主要为其他实现子类提供一些"公共设施"。它下面主要有两个子类，DefaultTransactionStatus和SimpleTransactionStatus。其中，DefaultTransactionStatus是Spring事务框架内部使用的主要TransactionStatus实现类。Spring事务框架内的各个TransactionManager的实现，大都借助于DefaultTransactionStatus来记载事务状态信息。SimpleTransactionStatus在Spring框架内部的实现中没有使用到，目前来看，主要用于测试目的。

19.2.3 PlatformTransactionManager

PlatformTransactionManager是Spring事务抽象框架的核心组件，我们之前已经提过了它的定义以及作用，所以，这里我们将更多地关注整个PlatformTransactionManager的层次体系，以及针对不同数据访问技术的实现类。

PlatformTransactionManager的整个抽象体系基于Strategy模式，由PlatformTransactionManager对事务界定进行统一抽象，而具体的界定策略的实现则交由具体的实现类。下面我们先来看一下有哪些可供我们使用的实现类。

1. PlatformTransactionManager实现类概览

PlatformTransactionManager的实现类可以划分为面向局部事务和面向全局事务两个分支。

面向局部事务的PlatformTransactionManager实现类。Spring为各种数据访问技术提供了现成的PlatformTransactionManager实现支持。表19-1给出了各种数据访问技术与它们对应的实现类的

关系。

表19-1 数据访问技术与PlatformTransactionManager实现类对应关系

数据访问技术	PlatformTransactionManager实现类
JDBC/iBATIS	DataSourceTransactionManager
Hibernate	HibernateTransactionManager
JDO	JdoTransactionManager
JPA（Java Persistence API）	JpaTransactionManager
TopLink	TopLinkTransactionManager
JMS	JmsTransactionManager
JCA Local Transaction	CciLocalTransactionManager

在这些实现类中，CciLocalTransactionManager可能是比较少见的实现，CCI是Common Client Interface的缩写。CciLocalTransactionManager主要是面向JCA的局部事务，本书不打算对JCA的集成做过多的阐述。如果你在实际项目中需要使用到JCA进行EIS（Enterprise Information System）系统集成，那么可以从Spring的参考文档获得使用Spring提供的JCA集成支持的相关信息。

有了这些实现类，我们在使用Spring的事务抽象框架进行事务管理的时候，只需要根据当前使用的数据访问技术，选择对应的PlatformTransactionManager实现类即可。

> **提示** 如果我们的应用程序需要同时使用Hibernate以及JDBC（或者iBATIS）进行数据访问，那么可以使用HibernateTransactionManager对基于Hibernate和JDBC（或者iBATIS）的事务进行统一管理，只要Hibernate的SessionFactory和JDBC（或者iBATIS）引用的是同一个DataSource就行。能猜到为什么吗？

面向全局事务的PlatformTransactionManager实现类。org.springframework.transaction.jta.JtaTransactionManager是Spring提供的支持分布式事务的PlatformTransac- tionManager实现。直接使用JTA规范接口进行分布式事务管理有以下几个问题。

- UserTransaction接口使用复杂不说（一长串的异常处理我们之前也见过了），它所公开的事务管理能力有限，对于事务的挂起（Suspend）以及恢复（Resume）操作，只有JTA的TransactionManager才支持。
- JTA规范并没有明确要求对TransactionManager的支持，这就造成虽然当下各个JTA提供商提供了TransactionManager的实现，但在应用服务器中公开的位置各有差异。为了进一步支持REQUIRES_NEW和NOT_SUPPORTED之类需要事务挂起以及恢复操作的事务传播行为，我们需要通过不同的方式来获取不同JTA提供商公开的TransactionManager实现。

鉴于此，JtaTransactionManager对各种JTA实现提供的分布式事务支持进行了统一封装，只不过它的所有的事务管理操作，最终都会委派给具体的JTA实现来完成。

对于典型的基于JTA的分布式事务管理，我们直接使用JtaTransactionManager就可以了。但某些时候，如果需要使用到各JTA产品的TransactionManager的特性，我们就可以为JtaTransactionManager注入这些JTA产品的javax.transaction.TransactionManager的实现。而至于说我们是通过应用服务器获取该TransactionManager，还是直接使用本地定义的TransactionManager（比如JOTM或者Atomikos等独立JTA实现产品的TransactionManager），则需要完全根

据当时的场景来决定了。能够为JtaTransactionManager提供具体的TransactionManager实现，为我们扩展JtaTransactionManager提供了很好的一个切入点。

JtaTransactionManager有两个子类OC4JJtaTransactionManager和WebLogicJtaTransactionManager，分别面向基于Oracle OC4J和Weglogic的JTA分布式事务管理。在这些情况下，需要使用具体的子类来代替通常的JtaTransactionManager。不过，大多数情况下，使用Spring提供的FactoryBean机制来获取不同JTA提供商提供的TransactionManager实现，然后注入JtaTransactionManager使用，是比较好的做法。org.springframework.transaction.jta包下，Spring提供了面向JOTM、Weblogic和Websphere的TransactionManager查找FactoryBean实现。如果需要，我们也可以根据情况，实现并给出其他的TransactionManager实现对应的用于查找的FactoryBean。

有了Spring的事务抽象框架，事务管理策略的转换也变得很简单，通常也只是简单的配置文件变更而已。如果我们最初只需要处理单一资源的事务管理，那么局部场景中的面向不同数据访问技术的PlatformTransactionManager实现，将是我们的最佳选择。即使后来需要引入分布式资源的事务管理，对于我们来说，也只是从局部事务场景中的某个PlatformTransactionManager实现转向JtaTransactionManager的变动而已。无论是编程式注入还是通过Spring的IoC容器注入，对于应用程序来说都不会造成很大的冲击。

2. 窥一斑而知全豹

PlatformTransactionManager的各个子类在实现时，基本上遵循统一的结构和理念。所以，我们不妨选择以DataSourceTransactionManager这一实现类作为切入点，以管中窥豹之势，一探Spring的抽象事务框架中各个PlatformTransactionManager实现类的奥秘所在。

不过，在开始之前，我们有必要先了解如下几个概念。

- transaction object。transaction object承载了当前事务的必要信息，PlatformTransactionManager实现类可以根据transaction object所提供的信息来决定如何处理当前事务。transaction object的概念类似于JTA规范中的javax.transaction.Transaction定义。
- TransactionSynchronization。TransactionSynchronization是可以注册到事务处理过程中的回调接口。它就像是事务处理的事件监听器，当事务处理的某些规定时点发生时，会调用TransactionSynchronization上的一些方法来执行相应的回调逻辑，如在事务完成后清理相应的系统资源等操作。Spring事务抽象框架所定义的TransactionSynchronization类似于JTA规范的javax.transaction.Synchronization，但比JTA的Synchronization提供了更多的回调方法，允许我们对事务的处理添加更多的回调逻辑。
- TransactionSynchronizationManager。类似于JTA规范中的javax.transaction.TransactionSynchronizationRegistry，我们通过TransactionSynchronizationManager来管理TransactionSynchronization、当前事务状态以及具体的事务资源。在介绍Spring事务框架实现原理的原型中，我们提到会将具体的事务资源，比如java.sql.Connection或者Hibernate Session绑定到线程，TransactionSynchronizationManager就是这些资源绑定的目的地。当然，从该类的名字也可以看出，它更多关注与事务相关的Synchronization的管理。

将它们与JTA规范中定义的接口相提并论，是因为这些概念只限于局部场景中对应的PlatformTransactionManager实现类使用，而JtaTransactionManager直接就使用对应的JTA产品提供的对应设施了，JtaTransactionManager的最终工作都是委派给具体的JTA实现产品，记得吗？

好的，有了这些铺垫，我们开始进入正题……

Spring的事务抽象框架以PlatformTransactionManager作为顶层抽象接口，具体的实现交给不

同的实现类，使用对象可以根据当前场景，选择使用或者替换哪一个具体的实现类。从这个层次看（见图19-10），整个框架的设计是以Strategy模式为基础的。不过，从各个实现类的继承层次上来看，Spring事务框架的实现则更多地依赖于模板方法模式。

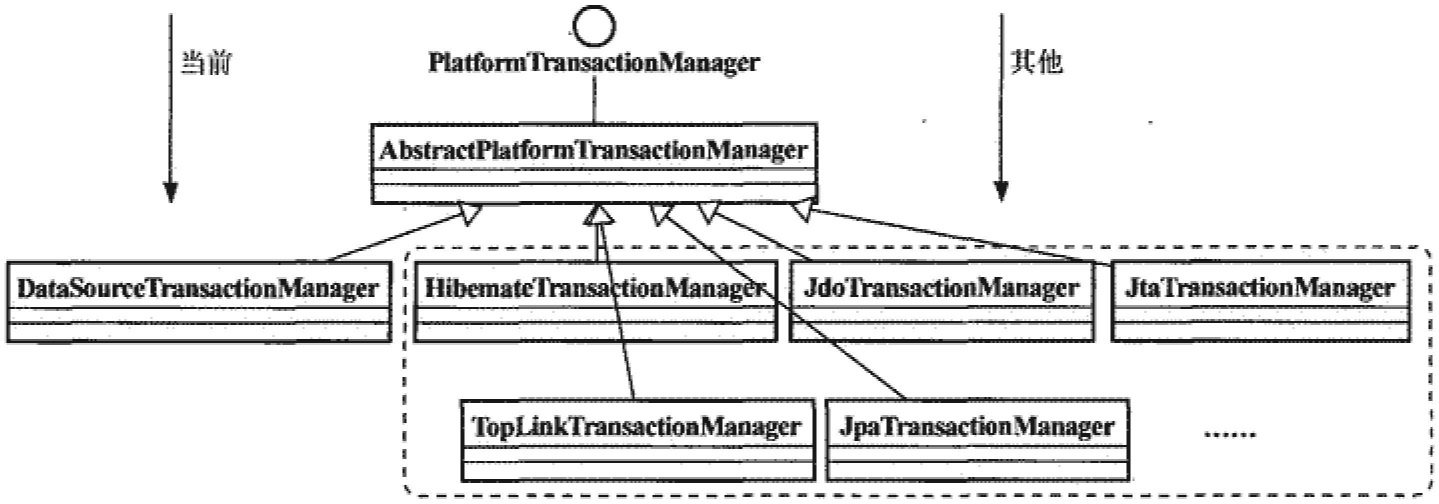

图19-10 DataSourceTransactionManager的实现层次

org.springframework.transaction.support.AbstractPlatformTransactionManager作为DataSourceTransactionManager的父类，以模板方法的形式封装了固定的事务处理逻辑，而只将与事务资源相关的操作以protected或者abstract方法的形式留给DataSourceTransactionManager来实现。作为模板方法父类，AbstractPlatformTransactionManager替各子类实现了以下固定的事务内部处理逻辑：

- 判定是否存在当前事务，然后根据判断结果执行不同的处理逻辑；
- 结合是否存在当前事务的情况，根据TransactionDefinition中指定的传播行为的不同语义执行后继逻辑；
- 根据情况挂起或者恢复事务；
- 提交事务之前检查readOnly字段是否被设置，如果是的话，以事务的回滚代替事务的提交；
- 在事务回滚的情况下，清理并恢复事务状态；
- 如果事务的Synchonization处于active状态，在事务处理的规定时点触发注册的Synchonization回调接口。

这些固定的事务内部处理逻辑大都体现在以下几个主要的模板方法中：

- public final TransactionStatus getTransaction(TransactionDefinitiondefinition) throws TransactionException
- public final void rollback(TransactionStatus status) throws Transaction-Exception
- public final void commit(TransactionStatus status) throws Transaction-Exception
- protected final SuspendedResourcesHolder suspend(Object transaction) throws TransactionException
- protected final void resume(Object transaction, SuspendedResourcesHolder resourcesHolder) throws TransactionException

我们可不打算把这几个模板方法都讲一遍。毕竟，只要了解了前面两三个模板方法的流程，整个事务处理的图景基本上就可以展现在我们眼前了。

我们先从第一个模板方法getTransaction(TransactionDefinition)开始。getTransaction(TransactionDefinition)的主要目的是开启一个事务，但需要在此之前判断一下之前是否存在一个事务。如果存在，则根据TransactionDefinition中的传播行为决定是挂起当前事务还是抛出异常。同样的，不存在事务的情况下，也需要根据传播行为的具体语义来决定如何处理。

getTransaction(TransactionDefinition)方法的处理逻辑，基本上按照下面的流程执行①。

(1) 获取transaction object，以判断是否存在当前事务。

```
Object transaction = doGetTransaction();
```

以上这行代码有如下两点需要申明。

- 获取的transaction object类型会因具体实现类的不同而各异，DataSourceTransactionManager会返回DataSourceTransactionManager.DataSourceTransactionObject类型实例，HibernateTransactionManager会返回HibernateTransactionManager.HibernateTransactionObject类型的实例，等等。AbstractPlatformTransactionManager不需要知道具体实现类返回的transaction object具体类型是什么，因为最终对transaction object的依赖都将通过方法参数进行传递，只要具体的实现类在取得transaction object参数后知道如何转型就行，所以，这一步返回的transaction为Object类型。
- doGetTransaction()是getTransaction(TransactionDefinition)模板方法公开给子类来实现的abstract类型方法。DataSourceTransactionManager在实现doGetTransaction()方法逻辑的时候，会从TransactionSynchronizationManager获取绑定的资源，然后添加到DataSourceTransactionObject之后返回。以此类推，其他AbstractPlatformTransactionManager子类都采用类似的逻辑实现了doGetTransaction()方法。

(2) 获取Log类的debug信息，避免之后的代码重复。如以下代码所示：

```
boolean debugEnabled = logger.isDebugEnabled();
if (debugEnabled) {
    logger.debug("Using transaction object [" + transaction + "]");
}
```

debugEnabled将以方法参数的形式在各方法调用间传递，以避免每次都调用logger.isDebugEnabled()获取debug日志状态。这一步与具体的事务处理流程关系不大。

(3) 检查TransactionDefinition参数合法性。如以下代码所示：

```
if (definition == null) {
    // Use defaults if no transaction definition given.
    definition = new DefaultTransactionDefinition();
}
```

如果definition参数为空，则创建一个DefaultTransactionDefinition实例以提供默认的事务定义数据。

(4) 根据先前获得的transaction object判断是否存在当前事务，根据判定结果采取不同的处理方式。如以下代码所示：

① 因为UML的Sequence图对于条件判断等逻辑无法以恰当的方式表示，而Activity图整个表示下来过大，不便显示，所以，最后决定以文字描述的方式进行。

```
if (isExistingTransaction(transaction)) {
    // Existing transaction found -> check propagation behavior to find out how to behave.
    return handleExistingTransaction(definition, transaction, debugEnabled);
}
```

默认情况下，isExistingTransaction(transaction)返回false，该方法需要具体子类根据情况进行覆写。对于DataSourceTransactionManager来说，它会根据传入的transaction所记载的信息进行判断，如下所示：

```
DataSourceTransactionObject txObject = (DataSourceTransactionObject) transaction;
return (txObject.getConnectionHolder() != null && txObject.getConnectionHolder().➥
isTransactionActive());
```

对于HibernateTransactionManager来说，则会将transaction强制转型为HibernateTransactionObject，然后根据HibernateTransactionObject所记载的信息来判断之前是否存在一个事务。其他具体实现类对isExistingTransaction(transaction)的处理亦是如此。

不管isExistingTransaction(transaction)返回结果如何，下面的处理主体上都是以TransactionDefinition中的传播行为为中心进行的。比如同样是PROPAGATION_REQUIRED，在存在当前事务与不存在当前事务两种情况下的处理是不同的，前者会使用之前的事务，后者则会创建新的事务，其他的传播行为的处理也是按照不同的场景分别处理。

(a) 如果isExistingTransaction(transaction)方法返回true，即存在当前事务的情况下，由handleExistingTransaction()方法统一处理存在当前事务，应该如何创建事务对应的TransactionStatus实例并返回。

如果definition定义的传播行为是PROPAGATION_NEVER，抛出异常并退出。如下：

```
if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION_NEVER) {
    throw new IllegalTransactionStateException(➥
    "Existing transaction found for transaction marked with propagation 'never'");
}
```

这是由TransactionDefinition.PROPAGATION_NEVER的语义决定的。

如果definition定义的传播行为是PROPAGATION_NOT_SUPPORTED，则挂起当前事务，然后返回。如下：

```
if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION_NOT_ ➥
SUPPORTED) {
    if (debugEnabled) {
        logger.debug("Suspending current transaction");
    }
    Object suspendedResources = suspend(transaction);
    boolean newSynchronization = (getTransactionSynchronization() == SYNCHRONIZATION_ ➥
    ALWAYS);
    return newTransactionStatus(
            definition, null, false, newSynchronization, debugEnabled, suspended- ➥
            Resources);
}
```

newTransactionStatus()方法返回一个DefaultTransactionStatus实例，因为我们挂起了当前事务。而PROPAGATION_NOT_SUPPORTED不需要事务，所以，返回的DefaultTransactionStatus不包含transaction object的信息（构造方法第二个参数）。

如果definition定义的传播行为是PROPAGATION_REQUIRES_NEW，则同样挂起当前事务，并开始一个新的事务并返回。如下：

```
if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION_REQUIRES_NEW) {
    if (debugEnabled) {
        logger.debug("Suspending current transaction, creating new transaction with name [" + ➥
        definition.getName() + "]");
    }
    SuspendedResourcesHolder suspendedResources = suspend(transaction);
    try {
        doBegin(transaction, definition);
    }
    catch (TransactionException beginEx) {
        try {
            resume(transaction, suspendedResources);
        }
        catch (TransactionException resumeEx) {
            logger.error( ➥
            "Inner transaction begin exception overridden by outer transaction ➥
            resume exception", beginEx);
            throw resumeEx;
        }
        throw beginEx;
    }
    boolean newSynchronization = (getTransactionSynchronization() != SYNCHRONIZATION_ ➥
    NEVER);
    return newTransactionStatus(➥
    definition, transaction, true, newSynchronization, debugEnabled, ➥
    suspendedResources);
}
```

AbstractPlatformTransactionManager首先将当前事务挂起，然后调用doBegin()方法开始新的事务。如果开始事务过程中出现异常，那么恢复之前挂起的事务。doBegin(transaction, definition)方法为abstract方法，需要具体子类来实现。在DataSourceTransactionManager中，doBegin()方法会首先检查传入的transaction，以提取必要信息判断之前是否存在绑定的connection信息。如果没有，则从DataSource中获取新的connection，然后将其AutoCommit状态改为false，并绑定到TransactionSynchronizationManager。当然，这期间也会涉及事务定义的应用以及条件检查等逻辑。在所有一切搞定之后，newTransactionStatus会创建一个包含definition、transaction object以及挂起的事务信息和其他状态信息的DefaultTransactionStatus实例并返回。

如果definition定义的传播行为是PROPAGATION_NESTED，根据情况创建嵌套事务，如通过Savepoint或者JTA的TransactionManager。如下：

```
if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION_NESTED) {
    if (!isNestedTransactionAllowed()) {
        throw new NestedTransactionNotSupportedException( ➥
        "Transaction manager does not allow nested transactions by default - " + ➥
        "specify 'nestedTransactionAllowed' property with value 'true'");
    }
    if (debugEnabled) {
        logger.debug("Creating nested transaction with name [" + definition.getName() + "]");
    }
    if (useSavepointForNestedTransaction()) {
        // Create savepoint within existing Spring-managed transaction,
        // through the SavepointManager API implemented by TransactionStatus.
        // Usually uses JDBC 3.0 savepoints. Never activates Spring synchronization.
        DefaultTransactionStatus status =
                newTransactionStatus(definition, transaction, false, false, debugEnabled, null);
```

```
            status.createAndHoldSavepoint();
            return status;
        }
        else {
            // Nested transaction through nested begin and commit/rollback calls.
            // Usually only for JTA: Spring synchronization might get activated here
            // in case of a pre-existing JTA transaction.
            doBegin(transaction, definition);
            boolean newSynchronization = (getTransactionSynchronization() != ➥
            SYNCHRONIZATION_NEVER);
            return newTransactionStatus(definition, transaction, true, newSynchronization, ➥
            debugEnabled, null);
        }
}
```

在这种情况下，会首先通过isNestedTransactionAllowed()方法检查AbstractPlatformTransactionManager的nestedTransactionAllowed属性状态。如果允许嵌套事务，那么还得分两种情况处理。对于DataSourceTransactionManager来说，因为它支持使用Savepoint创建嵌套事务，所以，会使用TransactionStatus创建相应的Savepoint并返回。而JtaTransactionManager则要依赖于具体JTA产品的TransactionManager提供嵌套事务支持。

useSavepointForNestedTransaction()方法默认返回true，即默认使用Savepoint创建嵌套事务。如果具体子类不支持使用Savepoint创建嵌套事务，则需要覆写该方法，如JtaTransactionManager。

如果需要检查事务状态匹配情况，则对当前存在事务与传入的defintion中定义的隔离级别与ReadOnly属性进行检查，如果数据不吻合，则抛出异常。如下：

```
if (isValidateExistingTransaction()) {
    // validate isolation
    // validate read only
    ...
}
```

AbstractPlatformTransactionManager的validateExistingTransaction属性默认值为false。如果你想进一步加强事务属性之间的一致性，可以将validateExistingTransaction属性设置为true，那么这时，以上代码即会被触发执行。

剩下的就是在其他情况下，直接构建TransactionStatus返回。比如对应PROPAGATION_SUPPORTS和PROPAGATION_REQUIRED的情况。

(b) 如果isExistingTransaction(transaction)方法返回false，即不存在当前事务的情况下。

当definition中定义的传播行为是PROPAGATION_MANDATORY的时候，抛出异常。因为不存在当前事务，所以根据PROPAGATION_MANDATORY的语义，理当如此。

当definition中定义的传播行为是PROPAGATION_REQUIRED、PROPAGATION_REQUIRES_NEW或者PROPAGATION_NESTED的时候，开启新的事务。如下：

```
if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION_REQUIRED ||➥
definition.getPropagationBehavior() == TransactionDefinition. ➥
PROPAGATION_REQUIRES_NEW ||➥
definition.getPropagationBehavior() == TransactionDefinition. ➥
PROPAGATION_NESTED) {
    SuspendedResourcesHolder suspendedResources = suspend(null);
    if (debugEnabled) {
        logger.debug("Creating new transaction with name [" + definition.getName() + "]: "➥
        + definition);
    }
```

```
        try {
            doBegin(transaction, definition);
        }
        catch (TransactionException ex) {
            resume(null, suspendedResources);
            throw ex;
        }
        boolean newSynchronization = (getTransactionSynchronization() != SYNCHRONIZATION_➥
        NEVER);
        return newTransactionStatus( ➥
        definition, transaction, true, newSynchronization, debugEnabled, ➥
        suspendedResources);
    }
```

之所以在doBegin之前先调用传入null的suspend()方法，是因为考虑到如果有注册的Synchronization，需要暂时将这些与将要开启的新事务无关的Synchronization先放一边。

剩下的其他情况，则返回不包含任何transaction object的TransactionStatus。这种情况下虽然是空的事务，但有可能需要处理在事务过程中相关的Synchronization。

从getTransaction(TransactionDefinition)的逻辑可以看出，AbstractPlatformTransactionManager更多关注的是事务处理过程中的总体逻辑，而跟具体事务资源相关的处理则交给了具体的子类来实现。

事务处理的完成有两种情况，即回滚事务或者提交事务，AbstractPlatformTransactionManager提供的rollback(TransactionStatus)和commit(TransactionStatus)两个模板方法，分别对应这两种情况下的处理。因为事务提交过程中可能需要处理回滚逻辑，我们不妨以commit(TransactionStatus)的实现流程看一下AbstractPlatformTransactionManager是如何处理事务完成的。

(1) 因为在事务处理过程中，我们可以通过TransactionStatus的setRollbackOnly()方法标记事务回滚，所以，commit(TransactionStatus)在具体提交事务之前会检查rollBackOnly状态。如果该状态被设置，那么转而执行事务的回滚操作。

rollback(TransactionStatus)的逻辑主要包含如下3点。

- 回滚事务。这里的回滚事务又可分为如下3种情况。
  - 如果是嵌套事务，则通过TransactionStatus释放Savepoint。
  - 如果TransactionStatus表示当前事务是一个新的事务，则调用子类的doRollback(TransactionStatus)方法真正的回滚事务。doRollback(TransactionStatus)是抽象方法，具体子类必须实现它。DataSourceTransactionManager在实现该方法的时候，无疑是调用connection.rollback()。HibernateTransactionManager会通过它Session上的Transaction的rollback()方法回滚事务。其他子类对doRollback(TransactionStatus)的实现逻辑依此类推。
  - 如果当前存在事务，并且rollbackOnly状态被设置，则调用由子类实现的doSetRollbackOnly(TransactionStatus)方法，各子类实现通常会将当前的transaction object的状态设置为rollBackOnly。
- **触发Synchronization事件。**

  回滚时触发的事件比提交时触发的事件要少，只有triggerBeforeCompletion (status)和triggerAfterCompletion()。
- 清理事务资源。如下所述。

- 设置TransactionStatus中的completed为完成状态。
- 清理与当前事务相关的Synchronization。
- 调用doCleanupAfterCompletion()释放事务资源，并解除到Transaction SynchronizationManager的资源绑定。对于DataSourceTransactionManager来说，是关闭数据库连接，然后解除对DataSource对应资源的绑定。
- 如果之前有挂起的事务，恢复挂起的事务。

(2) 如果rollBackOnly状态没被设置，则执行正常的事务提交操作。

commit(TransactionStatus)方法的其他逻辑与rollback(TransactionStatus)方法基本相似，只是几个具体操作有所差别，如下所述。

- 回滚事务现在变成是提交事务。提交事务的时候，也会涉及如下几种情况。
  - 决定是否提前检测全局的rollBackOnly标志。如果最外层事务已经被标记为rollBackOnly，并且failEarlyOnGlobalRollbackOnly为true，则抛出异常(如代码清单19-5所示)。
  - 如果提交事务之前发现TransactionStatus持有Savepoint，则释放它。这实际上是在处理嵌套事务的提交。
  - 如果TransactionStatus表示要提交的事务是一个新的事务，则调用子类的doCommit(TransactionStatus)方法实现提交事务。doCommit(TransactionStatus)也是AbstractPlatformTransactionManager公开给子类实现的抽象方法，子类必须实现该方法。对于DataSourceTransactionManager来说，因为事务的提交由Connection决定，所以会直接调用connection.commit()提交事务。其他的子类也会使用自身的局部事务API在该方法中实现事务的提交。
- 需要触发Synchronization相关事件。不过，触发的事件比rollback(TransactionStatus)中的要多，包括triggerBeforeCommit()、triggerBeforeCompletion()、triggerAfterCommit()和triggerAfterCompletion()。
- 如果AbstractPlatformTransactionManager的rollbackOnCommitFailure状态被设置为true，则表示如果在事务提交过程中出现异常，需要回滚事务。所以，当commit(TransactionStatus)方法捕获相应异常，并且检测到该字段被设置的时候，需要回滚事务。rollbackOnCommitFailure的默认值是false，表示即使提交过程中出现异常，也不回滚事务。
- 既然commit(TransactionStatus)与rollback(TransactionStatus)一样，都是意味着事务的完成，那么也需要在最后进行事务资源清理的工作，具体内容可以参照rollback(TransactionStatus)部分。

代码清单19-5 DataSourceTransactionManager事务提交部分代码摘录

```
boolean globalRollbackOnly = false;
if (status.isNewTransaction() || isFailEarlyOnGlobalRollbackOnly()) {
    globalRollbackOnly = status.isGlobalRollbackOnly();
}
...
if (globalRollbackOnly) {
    throw new UnexpectedRollbackException( ➥
    "Transaction silently rolled back because it has been marked as rollback-only");
}
```

suspend和resume两个方法的逻辑更好理解了。前者会把TransactionSynchronizationManager上当前事务对应的Synchroniazation信息以及资源获取到SuspendedResourcesHolder中，

然后解除这些绑定。后者则会将SuspendedResourcesHolder中保持的信息重新绑定到Transaction-SynchronizationManager。

实际上，如果将AbstractPlatformTransactionManager中处理Synchroniaztion回调以及事务传播行为的逻辑剥离一下的话，就会发现，整个的逻辑流程就是本章开始部分展示的实现原型所表达的那样。

图19-11展示了AbstractPlatformTransactionManager需要子类实现或者覆写的方法。对于各个子类来说，无非就是根据自身需要管理的资源和事务管理API提供这些方法的实现而已。

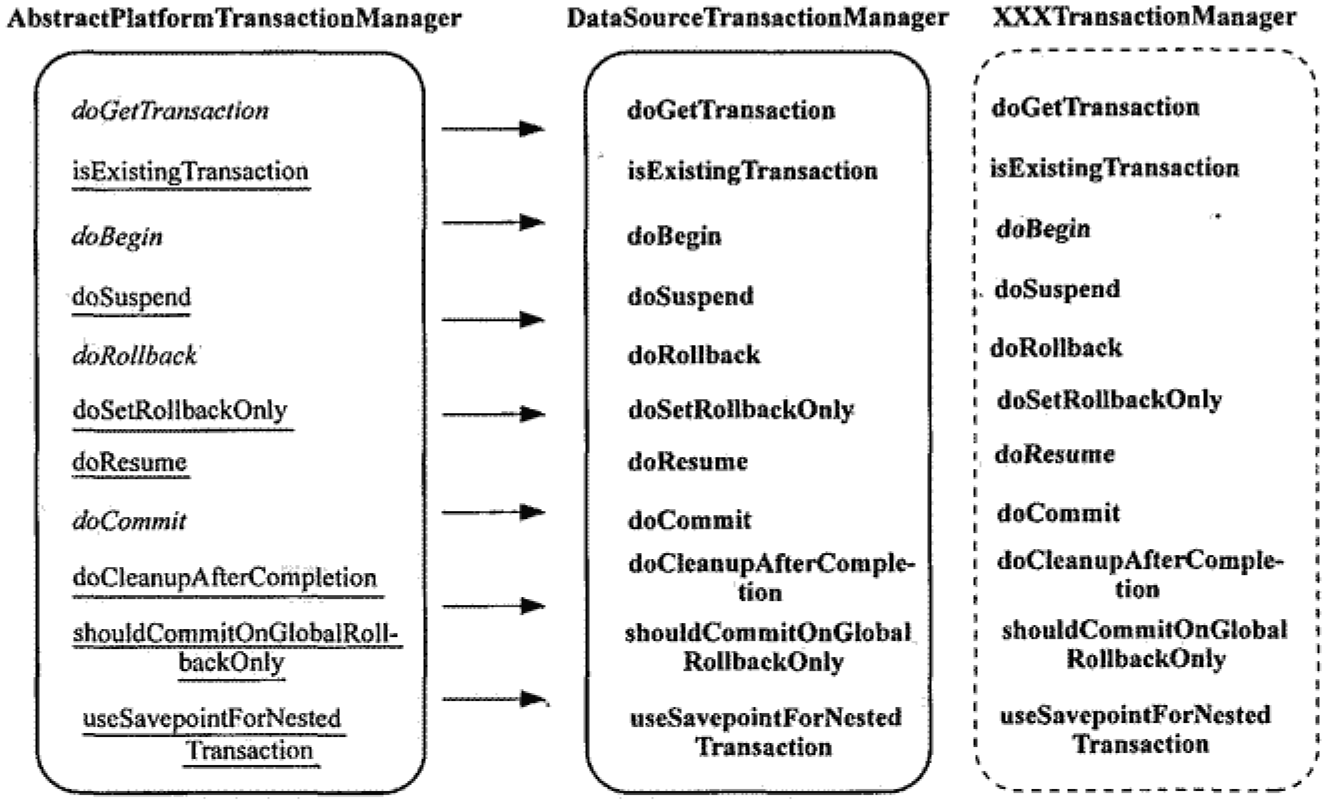

图19-11　模板类与实现类之间的纽带

19.3　小结

为了帮助你更容易地理解整个Spring事务管理抽象框架的设计和实现，我们在统一中原的过程中推导并实现了相应的PlatformTransactionManager以及相关原型实现。随后，在原型的基础上，对Spring事务管理抽象框架中的主要类进行了介绍和分析，尤其是对DataSourceTransactionManager的实现做了剖析，以期能够“以点带面”地帮助大家更好地理解Spring事务管理抽象框架中各PlatformTransactionManager通用的实现原理。

在了解了以上内容之后，我们将一起了解一下，在实际开发过程中，如何使用Spring的事务管理抽象框架让我们的日常开发生活变得更加美好。

第20章

使用Spring进行事务管理

本章内容

- 编程式事务管理
- 声明式事务管理

事务管理的实施通常有两种方式，即编程式事务管理和声明式事务管理。对于这两种事务管理方式的支持，Spring事务框架可以说是青出于蓝而胜于蓝。

20.1 编程式事务管理

通过Spring进行编程式事务管理有两种方式，要么直接使用PlatformTransactionManager，要么使用更方便的TransactionTemplate。二者各有优缺点，但总体上来说，推荐使用TransactionTemplate进行编程式事务管理。

20.1.1 直接使用 PlatformTransactionManager 进行编程式事务管理

PlatformTransactionManager接口定义了事务界定的基本操作，我们可以直接使用PlatformTransactionManager进行编程式事务管理，如代码清单20-1所示。

代码清单20-1 直接使用PlatformTransactionManager进行编程事务管理的代码示例

```
DefaultTransactionDefinition definition = new DefaultTransactionDefinition();
definition.setTimeout(20);
...
TransactionStatus txStatus = transactionManager.getTransaction(definition);
try
{
    // 业务逻辑实现
}
catch(ApplicationException e)
{
    transactionManager.rollback(txStatus);
    throw e;
}
catch(RuntimeException e)
{
    transactionManager.rollback(txStatus);
    throw e;
}
catch(Error e)
{
    transactionManager.rollback(txStatus);
    throw e;
```

```
}
transactionManager.commit(txStatus);
```

只要为transactionManager提供合适的PlatformTransactionManager实现，然后结合TransactionDefinition开启事务，并结合TransactionStatus来回滚或者提交事务，就可以完成针对当前对象的整个事务管理。

直接使用PlatformTransactionManager，我们可以完全控制整个事务处理过程，但是，缺点也是很明显的。从抽象事务操作以屏蔽不同事务管理API差异的角度看，PlatformTransactionManager可能已经足够了。但是，从应用程序开发的角度来看，却依然过于底层，只是期间的这些异常处理就够我们忙活的了。如果在每个需要事务管理的地方，全都使用PlatformTransactionManager进行事务管理，那么重复代码的数量将是惊人的。

鉴于使用PlatformTransactionManager进行事务管理的流程比较固定，各个事务管理期间只有部分逻辑存在差异，我们可以考虑像Spring的数据访问层那样，使用模板方法模式与Callback相结合的方式，对直接使用PlatformTransactionManager进行事务管理的代码进行封装。这就有了更方便的编程式事务管理方式，即使用TransactionTemplate的编程式事务管理。

20.1.2 使用 TransactionTemplate 进行编程式事务管理

org.springframework.transaction.support.TransactionTemplate对与PlatformTransactionManager相关的事务界定操作以及相关的异常处理进行了模板化封装，开发人员更多地关注于通过相应的Callback接口提供具体的事务界定内容即可。Spring针对TransactionTemplate提供了两个Callback接口，TransactionCallback和TransactionCallbackWithoutResult，二者的唯一区别就是是否需要返回执行结果。

使用TransactionTemplate进行事务管理的代码，看起来要比直接使用PlatformTransactionManager要简洁并且容易管理得多，如代码清单20-2所示。

代码清单20-2 使用TransactionTemplate进行编程式事务管理代码示例

```
TransactionTemplate txTemplate = ...;

Object result = txTemplate.execute(new TransactionCallback(){

    public Object doInTransaction(TransactionStatus txStatus) {
        Object result = null;
        // 各种事务操作……
        return result;
    }});
或者
txTemplate.execute(new TransactionCallbackWithoutResult(){
    @Override
    protected void doInTransactionWithoutResult(TransactionStatus txStatus) {
        // 事务操作1
        // 事务操作2
        // ...
    }});
```

TransactionTemplate会捕捉TransactionCallback或者TransactionCallbackWithoutResult事务操作中抛出的unchecked exception并回滚事务，然后将unchecked exception抛给上层处理。所以，现在我们只需要处理特定于应用程序的异常即可，而不用像直接使用PlatformTransactionManager那样，对所有可能的异常都进行处理。

如果事务处理期间没有任何问题，TransactionTemplate最终会为我们提交事务，唯一需要我们干预的就只剩下某些情况下的事务回滚了。如果在TransactionCallback或者Transaction-CallbackWithoutResult的事务操作过程中需要让当前事务回滚而不是最终提交，一般来说，我们有如下两种方式。

- 抛出unchecked exception，TransactionTemplate会为我们处理事务的回滚。如果事务操作中可能抛出checked exception，那么就得在Callback内部捕获，然后转译为unchecked exception后抛出。代码清单20-3给出了针对这种情况的处理代码示例。

代码清单20-3 抛出unchecked exception以使事务回滚的代码示例

```
txTemplate.execute(new TransactionCallbackWithoutResult(){
    @Override
    protected void doInTransactionWithoutResult(TransactionStatus txStatus) {
        try
        {
            ...
        }
        catch(CheckedException e)
        {
            // 抛出特定的异常类型以避免一般情况下使用的RuntimeException
            throw new RuntimeException(e);
        }
    }});
```

- 使用Callback接口公开的TransactionStatus将事务标记为rollBackOnly。Transaction-Template在最终提交事务的时候，如果检测到rollBackOnly标志状态被设置，将把提交事务改为回滚事务。代码清单20-4演示了这种情况。

代码清单20-4 设置rollBackOnly标志以使事务回滚的代码示例

```
txTemplate.execute(new TransactionCallbackWithoutResult(){
    @Override
    protected void doInTransactionWithoutResult(TransactionStatus txStatus) {
        boolean needRollback = false;
        ...

        if(needRollback)
            txStatus.setRollbackOnly();
    }});
```

对于事务操作中可能抛出的checked exception，如果既想回滚事务，又不想让它以unchecked exception的形式向上层传播的话，我们当然不能通过将其转译成unchecked exception的方式来处理。我们现在却可以通过TransactionStatus设置rollBackOnly来达到以上“一箭双雕”的目的（见代码清单20-5）。

代码清单20-5 捕获checked exception并回滚事务的代码示例

```
txTemplate.execute(new TransactionCallbackWithoutResult(){
    @Override
    protected void doInTransactionWithoutResult(TransactionStatus txStatus) {
        try
        {
            ...
        }
        catch(CheckedException e)
```

```
        {
            logger.warn("Tranaction is Rolled back!",e);
            txStatus.setRollbackOnly();
        }
    }});
```

这种情况下，需要注意一个问题，千万不要只txStatus.setRollbackOnly()而忘记记录日志。虽然你可能知道“swallow exception”是不对的，但这个地方确实容易忽略日志的记录，从而造成事务回滚了，而我们却不知道的情况。当发现数据库中本来应该删除的数据却依然存在，并且日志中也没有任何异常信息的时候，你就得花很长时间来想到底哪里出了问题了。难道是程序没运行？才怪！

虽然使用TransactionTemplate要比直接使用PlatformTransactionManager更加便捷，但TransactionTemplate无法处理当事务操作中需要向上层抛出原来的checked exception的情况。你应该也发现了，实际上TransactionCallback或者TransactionCallbackWithoutResult的方法定义中没有声明抛出任何checked exception，直接使用PlatformTransactionManager则没有这样的限制。不过，这应该并不会过多地限制TransactionTemplate展现其“个人魅力”吧？

20.1.3　编程创建基于 Savepoint 的嵌套事务

TransactionStatus不但可以在事务处理期间通过setRollbackOnly()方法来干预事务的状态，如果需要，作为SavepointManager，它也可以帮助我们使用Savepoint机制来创建嵌套事务。

以银行账户间转账为例，来说明如何使用TransactionStatus创建基于Savepoint的嵌套事务。现在不是从一个账户转到另一个账户，而是从一个账户转到两个账户，一个是主账户，一个备用账户。如果向主账户转账失败，则将金额转入备用账户。总之，金额从第一个账户取出之后，必须存入两个账户的其中一个，以保证整个事务的完整性。在这样的前提下，我们的事务管理代码基本上如代码清单20-6所示。

代码清单20-6　使用Savepoint创建嵌套事务代码示例

```
txTemplate.execute(new TransactionCallbackWithoutResult(){
    @Override
    protected void doInTransactionWithoutResult(TransactionStatus txStatus) {
        BigDecimal transferAmount = new BigDecimal("20000");
        try
        {
            withdraw("WITHDRAW_ACOUNT_ID",transferAmount);

            Object savePointBeforeDeposit = txStatus.createSavepoint();
            try
            {
                deposit("MAIN_ACOUNT_ID",transferAmount);
            }
            catch(DepositException ex)
            {
                logger.warn("rollback to savepoint for main acount transfer failure",ex);
                txStatus.rollbackToSavepoint(savePointBeforeDeposit);

                deposit("SECONDARY_ACOUNT_ID", transferAmount);
            }
            finally
            {
                txStatus.releaseSavepoint(savePointBeforeDeposit);
            }
        }
```

```
        catch(TransferException e)
        {
            logger.warn("failed to complete transfer operation!",e);
            txStatus.setRollbackOnly();
        }
    }});
```

当然，如果在转账期间的异常是unchecked exception，最外层的捕捉TransferException是没有太多必要的（这种情况下，TransactionTemplate将自动回滚事务）。

在这里，使用Savepoint创建嵌套事务的好处是，即使deposit过程中涉及多笔数据的更新，通过txStatus.rollbackToSavepoint(savePointBeforeDeposit)也可以将这些数据恢复到没有存入金额之前的状态，而不会破坏当前事务的完整性。如果在这里通过传播行为是PROPAGATION_REQUIRES_NEW的TransactionDefinition创建一个新的事务的话，虽然deposit过程中出现问题也可以回滚数据，但取款与存款的操作就不在同一个事务中了（取款在当前事务，存款在另一个新的事务），这无疑违反了事务的ACID属性。

> **注意** 使用TransactionStatus创建基于Savepoint的嵌套事务需要底层的PlatformTransactionManager实现类的支持，当前只有在JDBC 3.0驱动下的DataSourceTransactionManager可用。

通过使用TransactionStatus创建基于Savepoint的嵌套事务并非创建嵌套事务的唯一方式，也并非最方便的方式。实际上，我们更倾向于使用结合PROPAGATION_NESTED传播行为的声明式事务管理方式。

20.2 声明式事务管理

直接使用编程式事务管理的不足就是，事务管理代码与业务逻辑代码相互混杂。而声明式事务管理则可以避免这种不同系统关注点之间的纠缠，使得事务管理代码不用再去影响具体业务逻辑的实现。

20.2.1 引子

声明式事务实际上并没有想象中那么神秘，在我们将事务管理这一横切关注点从原来硬编码事务管理逻辑的系统中剥离出来后，就会发现，声明式事务已经在“幸福终点站”那里等着我们了。

先不管Spring是如何提供声明式事务管理的。如果要我们从原来硬编码事务管理的系统中，将这些事务管理相关的代码从业务对象中剥离出来，我们会怎么做？ 最土的一个办法是，为每个Service都提供一个TransactionFacade，将事务管理逻辑集中到TransactionFacade中，而Service实现类可以对事务管理一无所知，只要保证针对所有Service的调用必须走TransactionFacade即可。整个情形看起来就是图20-1所示的样子。

虽然这种方法可以实现事务管理代码与业务逻辑代码之间的分离，但是如果不做一些变通的话，在实际的开发中也不会给我们带来更多的好处。难道你想为每个Serivce对象都实现一个对应的TransactionFacade对象吗？

对，经过Spring AOP的洗礼，你应该已经想到了，动态代理不就是为这个设计的吗？呵呵，不过，原理归原理，要真的实现这种功能，直接使用Spring AOP才是正道啊。事务管理本身就是一种横切关注点，与其他的横切关注点本质上没有任何区别，所以，我们完全可以为其提供相应的Advice实现，然后织入到系统中需要该横切逻辑的Joinpoint处。这样不就达到了将事务管理逻辑从业务逻辑实现中

剥离出来的目的了吗？现在，我们要做的就是提供一个拦截器，在业务方法执行开始之前开启一个事务，当方法执行完成或者异常退出的时候就提交事务或者回滚事务。有了Spring的编程式事务管理API的支持，实现这样的一个拦截器对于我们来说应该就很容易了，代码清单20-7给出了这样一个拦截器原型的代码实现。

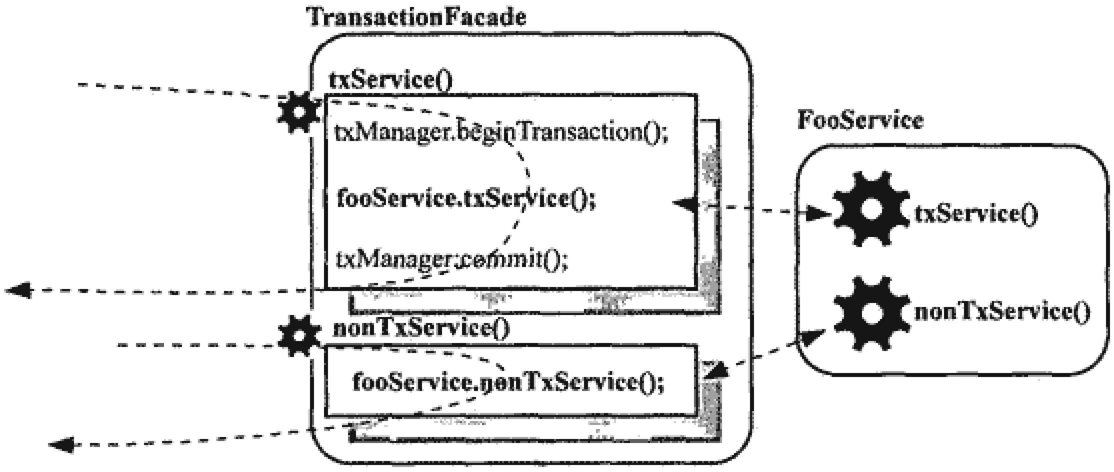

图20-1　TransactionFacade场景示意图

代码清单20-7　用于声明式事务管理的拦截器原型代码示例

```
public class PrototypeTransactionInterceptor implements MethodInterceptor
{
    private PlatformTransactionManager transactionManager;

    public Object invoke(MethodInvocation invocation) throws Throwable {
        Method method = invocation.getMethod();
        TransactionDefinition definition = getTransactionDefinitionByMethod(method);
        TransactionStatus txStatus = transactionManager.getTransaction(definition);
        Object result = null;
        try
        {
            result = invocation.proceed();
        }
        catch(Throwable t)
        {
            if(needRollbackOn(t))
            {
                transactionManager.rollback(txStatus);
            }
            else
            {
                transactionManager.commit(txStatus);
            }
            throw t;
        }
        transactionManager.commit(txStatus);
        return result;
    }
    protected boolean needRollbackOn(Throwable t) {
        // TODO ...更多实现细节
        return false;
    }
    protected TransactionDefinition getTransactionDefinitionByMethod(Method method) {
        // TODO ...更多实现细节
```

```
        return null;
    }
    public PlatformTransactionManager getTransactionManager() {
        return transactionManager;
    }
    public void setTransactionManager(PlatformTransactionManager transactionManager) {
        this.transactionManager = transactionManager;
    }
}
```


在实现这样的一个拦截器的过程中，我们会发现有些逻辑不好确定，如下所述。

- 针对每个对象业务方法的拦截，需要知道该方法是否需要事务支持。如果需要，针对该事务的TransactionDefinition相关信息又从哪里获得？
- 如果调用方法过程中抛出异常，如何对这些异常进行处理，哪些异常抛出的情况下需要回滚事务，哪些异常抛出的情况下又不需要？

我们需要提供某种方式来记载业务方法与对应的事务信息之间的映射关系，拦截器只要查询这种映射关系，就可以知道要不要为当前业务方法创建事务。如果要创建事务，则以业务方法作为标志，到映射关系中查找创建事务所需要的信息，然后创建事务。我们当然可以将这种映射信息写死到拦截器实现类中。不过，将它们放在外部配置文件（例如properties或者XML文件）或者Java源代码的注解中才是当下比较流行的方式。这种映射信息正规来讲，叫做驱动事务的元数据（Metadata）。有了类似于PrototypeTransactionInterceptor这样的AOP装备的支持，系统的事务管理就变成了只需要在元数据中声明相应的事务控制信息。

Spring提供了用于声明式事务管理的一切设施（使用org.springframework.transaction.interceptor.TransactionInterceptor实现替代PrototypeTransactionInterceptor），对于我们来说，所要做的只是决定要使用XML元数据驱动，还是使用注解元数据驱动的声明式事务管理。

> **提示** 什么？我还没有公布第二个问题是如何解决的？还记得在TransactionDefinition部分提到的用于声明式事务的TransactionAttribute定义吗？其实仅需要将“在什么异常类型抛出的情况下需要回滚事务”的信息补充到默认的TransactionDefintion定义中，用于事务处理的拦截器根据获得的TransactionDefinition提供的该信息，就能够决定出现异常的情况下应该如何结束事务。而TransactionAttribute就是增加了异常回滚信息的TransactionDefinition定义。用TransactionAttribute实现类代替默认的TransactionDefintion实现类创建事务不就结了？

20.2.2 XML 元数据驱动的声明式事务

Spring允许我们在IoC容器的配置文件中直接指定事务相关的元数据，从1.*x*版本发展到2.*x*版本，不管配置形式如何变化，所要达到的目的以及底层的机制却是不变的，配置形式的演化只是为了能够更加简洁方便。

从Spring 1.*x*到2.*x*，大体上来说，我们可以使用以下4种配置方式在IoC容器的配置文件中指定事务需要的元数据。

- 使用ProxyFactory（ProxyFactoryBean）+TransactionInterceptor。
- 使用“一站式”的TransactionProxyFactoryBean。
- 使用BeanNameAutoProxyCreator。

- 使用Spring 2.x的声明事务配置方式。

我们将对这4种配置方式给出详细的实例和相关内容的讲解。不过在此之前，我们需要先声明一个事务加之于上的模型。总不能将所有的配置都构建成“空中楼阁”不是？

总是FooService或者BarService，可能大家都看烦了，所以，我们这回打算换个口味。假设我们要搭建一个QuoteService，先暂且不管它是应用于一般的证券系统还是外汇系统，总之，通过它，我们能够查询报价信息，并且必要的话，也可以更新底层数据内容。现在的QuoteService不是一个远程服务，它的目的也很简单，基本上就是实现一个Quote信息的基本管理功能（第33章将介绍如何通过Spring将QuoteService以远程服务的方式公开出来）。

首先定义的是对应QuoteService的服务接口，面向接口编程为管理问题域提供了很好的抽象方式，接口定义如下所示：

```
public interface IQuoteService {
    Quote getQuate();
    Quote getQuateByDateTime(DateTime dateTime);
    void saveQuote(Quote quote);
    void updateQuote(Quote quote);
    void deleteQuote(Quote quote);
}
```

QuoteService实现了IQuoteService接口，其定义见代码清单20-8。

代码清单20-8　QuoteService定义

```
public class QuoteService implements IQuoteService
{
    private JdbcTemplate jdbcTemplate;

    public Quote getQuate() {
        return (Quote)getJdbcTemplate().queryForObject("", new RowMapper(){

            public Object mapRow(ResultSet rs, int row) throws SQLException {
                Quote quote = new Quote();
                // ...
                return quote;
            }});
    }
    public Quote getQuateByDateTime(DateTime dateTime) {
        throw new NotImplementedException();
    }
    public void saveQuote(Quote quote) {
        throw new NotImplementedException();
    }
    public void updateQuote(Quote quote) {
        throw new NotImplementedException();
    }
    public void deleteQuote(Quote quote) {
        throw new NotImplementedException();
    }
    public JdbcTemplate getJdbcTemplate() {
        return jdbcTemplate;
    }
    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }
}
```

这里只是为了说明针对QuoteService的事务管理，所以，暂且直接使用JdbcTemplate进行数据

访问。如果系统需要进一步的抽象，可以将数据访问逻辑提取到DAO层，屏蔽数据访问技术造成的差异。我们仅给出一个方法的实现，其余的先抛出NotImplementedException。当系统运行时，这可以告诉我们还有工作没有完成。

在有了QuoteService这个简单的模型之后，我们开始进入正题。

1. 使用ProxyFactory（ProxyFactoryBean）+TransactionInterceptor

XML元数据驱动的声明式事务，本质上来说，是为TransactionInterceptor提供需要的TransactionDefiniton（具体来说，是TransactionAttribute）信息。而至于说将TransactionInterceptor给出的事务管理相关的横切逻辑加到相应的业务对象上的过程，则纯粹就是一个AOP的配置过程。既然如此，最基本的方式当然就是，直接通过ProxyFactoryBean（或者ProxyFactory）完成事务管理这一横切关注点到系统的织入工作。所以，使用ProxyFactoryBean和TransactionInterceptor为我们的QuoteService添加声明式事务，看起来就是代码清单20-9所示的样子。

代码清单20-9 使用ProxyFactoryBean和TransactionInterceptor进行事务管理的配置示例

```
<beans>
    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" ➥
    destroy-method="close">
        <property name="url" value="jdbc:mysql://localhost/databaseName"/>
        <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
        <property name="username" value="..."/>
        <property name="password" value="..."/>
    </bean>

    <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <bean id="transactionManager" class="org.springframework.jdbc.datasource. ➥
    DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <bean id="transactionInterceptor" class="org.springframework.transaction.interceptor. ➥
    TransactionInterceptor">
        <property name="transactionManager" ref="transactionManager"/>
        <property name="transactionAttributes">
            <props>
                <prop key="getQuate*">PROPAGATION_SUPPORTS,readOnly,timeout_20</prop>
                <prop key="saveQuote">PROPAGATION_REQUIRED</prop>
                <prop key="updateQuote">PROPAGATION_REQUIRED</prop>
                <prop key="deleteQuote">PROPAGATION_REQUIRED</prop>
            </props>
        </property>
    </bean>

    <bean id="quoteServiceTarget" class="...QuoteService">
        <property name="jdbcTemplate" ref="jdbcTemplate"/>
    </bean>

    <bean id="quoteService" class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="target" ref="quoteServiceTarget"/>
        <property name="proxyInterfaces" value="...IQuoteService"/>
        <property name="interceptorNames">
            <list>
                <value>transactionInterceptor</value>
```

```
            </list>
        </property>
    </bean>

    <bean id="client" class="...QuoteSerivceClient">
        <property name="quoteService" ref="quoteService"/>
    </bean>
</beans>
```

一般意义上，QuoteSerivceClient是使用QuoteService的应用类，当我们通过它调用QuoteService的业务方法的时候，除了getQuote()方法之外，其他方法，比如getQuoteService().saveQuote(quote)，因为抛出了unchecked的NotImplementedException，从而导致事务自动回滚。这说明我们的声明式事务生效了，不是吗？如下所示：

```
    1094 [main] DEBUG org.springframework.transaction.support.TransactionSynchronization-
Manager  - Initializing transaction synchronization
    1094 [main] DEBUG org.springframework.transaction.interceptor.TransactionInterceptor-
Getting transaction for [...IQuoteService.saveQuote]
    1094 [main] DEBUG org.springframework.transaction.interceptor.TransactionInterceptor-
Completing transaction for [...IQuoteService.saveQuote] after exception:
org.apache.commons.lang.NotImplementedException: Code is not implemented
    1094 [main] DEBUG org.springframework.transaction.interceptor.RuleBasedTransaction-
Attribute  - Applying rules to determine whether transaction should rollback on
org.apache.commons.lang.NotImplementedException: Code is not implemented
    1094 [main] DEBUG org.springframework.transaction.interceptor.RuleBasedTransaction-
Attribute  - Winning rollback rule is: null
    1094 [main] DEBUG org.springframework.transaction.interceptor.RuleBasedTransaction-
Attribute  - No relevant rollback rule found: applying superclass default
    1094 [main] DEBUG org.springframework.jdbc.datasource.DataSourceTransactionManager-
Triggering beforeCompletion synchronization
    1094 [main] DEBUG org.springframework.jdbc.datasource.DataSourceTransactionManager-
Initiating transaction rollback
    1094 [main] DEBUG org.springframework.jdbc.datasource.DataSourceTransactionManager-
Rolling back JDBC transaction on Connection [org.apache.commons.dbcp.PoolableConnection@b76fa]
    1094 [main] DEBUG org.springframework.jdbc.datasource.DataSourceTransactionManager-
Triggering afterCompletion synchronization
    1094 [main] DEBUG org.springframework.transaction.support.TransactionSynchronization-
Manager  - Clearing transaction synchronization
```

现在让我们回头探究一下，在使用ProxyFactoryBean和TransactionInterceptor进行声明式事务的过程中，到底有哪些奥秘和需要我们注意的地方。

要使声明式事务起作用，需要几个方面协同工作，这包括事务要管理的具体数据资源类型、采用的数据访问技术、特定的事务管理器实现，以及TransactionInterceptor提供的业务方法拦截功能。具体到我们的QuoteService来说，我们要对数据库提供的数据资源进行事务操作，所以，配置中需要dataSource的定义，这是事务操作的基础，也就是如下所示配置内容：

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" ➥
destroy-method="close">
    <property name="url" value="jdbc:mysql://localhost/databaseName"/>
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="username" value="..."/>
    <property name="password" value="..."/>
</bean>
```

要对数据资源进行访问，QuoteService采用的是JDBC的方式，即直接使用Spring提供的JdbcTemplate，那么，我们需要提供JdbcTemplate以及特定于JDBC的事务管理器DataSource-

TransactionManager，如下定义：

```
<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
    <property name="dataSource" ref="dataSource"/>
</bean>

<bean id="transactionManager" class="org.springframework.jdbc.datasource. ➥
DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>
```


数据访问技术与事务管理器的类型是一一对应的，否则就有“驴唇不对马嘴”之嫌，让A公司的老板来管理B公司的事务，B公司的员工当然可以不理他。所以，如果你的QuoteService要用Hibernate进行数据访问，那么请提供HibernateTransactionManager作为事务管理器。如果你的QuoteService要用JDO进行数据访问，请将DataSourceTransactionManager替换为JdoTransactionManager，以此类推。

TransactionInterceptor是整个声明式事务的主体。要让它发挥事务管理的职能，需要为它提供一个事务管理器。要让它知道该为哪个方法加诸什么事务，是否要加诸事务，需要为它提供必要的映射信息，如代码清单20-10所示。

代码清单20-10 TransactionInterceptor配置示例

```
<bean id="transactionInterceptor" class="org.springframework.transaction.interceptor. ➥
TransactionInterceptor">
    <property name="transactionManager" ref="transactionManager"/>
    <property name="transactionAttributes">
        <props>
            <prop key="getQuate*">PROPAGATION_SUPPORTS,readOnly,timeout_20</prop>
            <prop key="saveQuote">PROPAGATION_REQUIRED</prop>
            <prop key="updateQuote">PROPAGATION_REQUIRED</prop>
            <prop key="deleteQuote">PROPAGATION_REQUIRED</prop>
        </props>
    </property>
</bean>
```

TransactionInterceptor所需要的业务对象上的每个业务方法的事务管理信息，通过org.springframework.transaction.interceptor.TransactionAttributeSource接口来获取，该接口定义如下：

```
public interface TransactionAttributeSource {
    TransactionAttribute getTransactionAttribute(Method method, Class targetClass);
}
```

通常，TransactionAttributeSource的具体实现类以不同的形式存储了从不同位置获取的事务管理元数据信息，比如NameMatchTransactionAttributeSource将以方法名作为映射信息的Key，对相应的元数据进行存储；MethodMapTransactionAttributeSource则直接以对应每个方法的Method实例作为Key，来存储元数据对应的映射信息；而AnnotationTransactionAttributeSource则是直接从源代码中的注解中获取对应每个业务方法的事务管理信息。

我们通常通过如下两个属性为TransactionInterceptor设置需要的TransactionAttributeSource。

- **transactionAttributes属性。**我们刚才就是使用transactionAttributes属性为TransactionInterceptor设置的映射信息，transactionAttributes是Properties类型，Transac-

tionInterceptor内部将通过transactionAttributes提供的信息构造一个NameMatchTransactionAttributeSource类型的TransactionAttributeSource使用。

- **transactionAttributeSource属性。** transactionAttributeSource属性就是直接可用的TransactionAttributeSource类型，但是我们在IoC容器的配置文件中指定的是String形式。所以，容器将通过org.springframework.transaction.interceptor.TransactionAttributeSourceEditor对String形式的值进行一个转换，再设置给TransactionInterceptor，转换后的具体TransactionAttributeSource实现类为MethodMapTransactionAttributeSource。同样的元数据，通过transactionAttributeSource设置如下：

```
<property name="transactionAttributeSource">
    <value>
    org.spring21.package.IQuoteService.getQuate*=PROPAGATION_SUPPORTS, ➥
    readOnly,timeout_20
    org.spring21.package.IQuoteService.saveQuote=PROPAGATION_REQUIRED
    org.spring21.package.IQuoteService.updateQuote=PROPAGATION_REQUIRED
    org.spring21.package.IQuoteService.deleteQuote=PROPAGATION_REQUIRED
    </value>
</property>
```

唯一需要注意的就是，现在需要指定全限定的类名和方法名。

> **注意**　也可以通过transactionAttributeSources属性指定元数据信息，它是transactionAttributeSource的复数形式。

在TransactionInterceptor准备就绪之后，剩下的就是通过Spring AOP将业务对象与其绑定，见代码清单20-11。

代码清单20-11　使用ProxyFactoryBean织入TransactionInterceptor的事务横切逻辑代码示例

```
<bean id="quoteServiceTarget" class="...QuoteService">
    <property name="jdbcTemplate" ref="jdbcTemplate"/>
</bean>

<bean id="quoteService" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target" ref="quoteServiceTarget"/>
    <property name="proxyInterfaces" value="...IQuoteService"/>
    <property name="interceptorNames">
        <list>
            <value>transactionInterceptor</value>
        </list>
    </property>
</bean>

<bean id="client" class="...QuoteSerivceClient">
    <property name="quoteService" ref="quoteService"/>
</bean>
```

只不过，一定不要把quoteServiceTarget当作quoteService而注入给需要的对象，否则，事务管理不生效那就怨不得别人了。

元数据中事务属性指定规则

以String形式给出的事务属性是有一定规则的，这个规则由org.springframework.transaction.interceptor.TransactionAttributeEditor类的逻辑来界定，TransactionAttributeEditor负责将String形式给出的事务属性转换为具体的TransactionAttribute实例（RuleBasedTransactionAttribute）。

String形式的事务属性规则如下：

```
PROPAGATION_NAME,[ISOLATION_NAME],[readOnly],[timeout_NNNN],[+Exception1],[-Exception2]
```

除了PROPAGATION_NAME是必须的之外，其他规则可以根据需要决定是否需要指定，各个区段之间以逗号分隔。

- PROPAGATION_NAME。对应org.springframework.transaction.TransactionDefinition中定义的与传播行为相关的常量名，即PROPAGATION_REQUIRED、PROPAGATION_REQUIRES_NEW、PROPAGATION_SUPPORTS、PROPAGATION_MANDATORY、PROPAGATION_NESTED、PROPAGATION_NOT_SUPPORTED和PROPAGATION_NEVER。
- ISOLATION_NAME。对应org.springframework.transaction.TransactionDefinition中定义的与隔离级别相关的常量名，即ISOLATION_DEFAULT、ISOLATION_READ_UNCOMMITTED、ISOLATION_READ_COMMITTED、ISOLATION_REPEATABLE_READ和ISOLATION_SERIALIZABLE。
- readOnly。如果需要将当前事务标明为只读事务，那么追加该值即可。
- timeout_NNNN。指定事务的超时时间，需要以timeout_为前缀，后面加数字的形式，如timeout_20限定超时时间为20秒。
- [+Exception1],[-Exception2]。自定义异常回滚规则。前缀加号（+）后跟具体的异常类型表示，即使业务方法抛出该异常也同样提交事务。前缀减号（-）后跟具体的异常类型表示，业务方法抛出该异常的时候回滚事务。比如，当业务方法抛出QuoteException的时候，我们想要事务回滚，则可以如下这样指定：

```
serviceMethod=PROPAGATION_REQUIRED,-QuoteException
```

因为“unchecked exception”默认情况下会自动回滚，所以，通过自定义异常回滚规则主要是指定“checked exception”类型的应用异常。

使用ProxyFactoryBean和TransactionInterceptor实现声明式事务，可以从最低层次上理解Spring提供的声明式事务是如何运作的。不过，为了减少配置量，进一步提高开发效率，我们会探索更加便捷的方式，TransactionProxyFactoryBean就向这一目标更迈进了一步。

2. 使用“一站式”的TransactionProxyFactoryBean

TransactionProxyFactoryBean是专门面向事务管理的ProxyFactoryBean，它直接将TransactionInterceptor纳入自身进行管理。使用TransactionProxyFactoryBean代替ProxyFactoryBean进行声明式事务管理，不需要单独声明TransactionInterceptor的bean定义，有关事务的元数据、事务管理器等信息，全都通过TransactionProxyFactoryBean的bean定义指定就可以。

这样，同样针对QuoteService的声明式事务管理，使用TransactionProxyFactoryBean后的样子如代码清单20-12所示。

代码清单20-12　使用TransactionProxyFactoryBean进行声明式事务管理配置代码示例

```
<beans>
    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method ➥
    ="close">
        <property name="url" value="jdbc:mysql://localhost/dbName"/>
        <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
        <property name="username" value="..."/>
        <property name="password" value="..."/>
    </bean>

    <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
    <property name="dataSource" ref="dataSource"/>
    </bean>

    <bean id="transactionManager" class="org.springframework.jdbc.datasource. ➥
    DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <bean id="quoteServiceTarget" class="...QuoteService">
        <property name="jdbcTemplate" ref="jdbcTemplate"/>
    </bean>

    <bean id="quoteService" class="org.springframework.transaction.interceptor. ➥
    TransactionProxyFactoryBean">
        <property name="target" ref="quoteServiceTarget"/>
        <property name="proxyInterfaces" value="...IQuoteService"/>
        <property name="transactionManager" ref="transactionManager"/>
        <property name="transactionAttributes">
            <props>
                <prop key="getQuate*">PROPAGATION_SUPPORTS,readOnly,timeout_10</prop>
                <prop key="saveQuote">PROPAGATION_REQUIRED</prop>
                <prop key="updateQuote">PROPAGATION_REQUIRED</prop>
                <prop key="deleteQuote">PROPAGATION_REQUIRED</prop>
            </props>
        </property>
    </bean>

    <bean id="client" class="...QuoteSerivceClient">
        <property name="quoteService" ref="quoteService"/>
    </bean>
</beans>
```

现在，TransactionProxyFactoryBean集ProxyFactoryBean、TransactionInterceptor功能于一身，一心一意地为声明式事务管理做贡献了。

不过，我们也看到了，针对TransactionProxyFactoryBean的bean定义看起来不是那么苗条，如果每个需要声明式事务的业务对象都来这么一下子，那么配置量可着实不轻松。所以，通常情况下，我们会使用bean定义模板的方式，来简化使用TransactionProxyFactoryBean进行声明式事务的配置，如代码清单20-13所示。

代码清单20-13　使用bean定义模板方式简化使用TransactionProxyFactoryBean的声明式事务配置代码示例

```
<bean id="txProxyFactoryBean" class="org.springframework.transaction.interceptor. ➥
TransactionProxyFactoryBean" abstract="true">
    <property name="proxyInterfaces" value="...IQuoteService"/>
    <property name="transactionManager" ref="transactionManager"/>
    <property name="transactionAttributes">
        <props>
```

```
            <prop key="getQuate*">PROPAGATION_SUPPORTS,readOnly,timeout_10</prop>
            <prop key="saveQuote">PROPAGATION_REQUIRED</prop>
            <prop key="updateQuote">PROPAGATION_REQUIRED</prop>
            <prop key="deleteQuote">PROPAGATION_REQUIRED</prop>
        </props>
    </property>
</bean>

<bean id="quoteService" parent="txProxyFactoryBean">
    <property name="target" ref="quoteServiceTarget"/>
</bean>

<bean id="quoteService2" parent="txProxyFactoryBean">
    <property name="target" ref="otherQuoteServiceTarget"/>
</bean>

...
```

将共有的一些属性提取到txProxyFactoryBean的bean定义模板中，就可以减少每次配置单独业务对象对应的bean定义的工作量。

相对于直接使用ProxyFactoryBean和TransactionInterceptor，使用TransactionProxyFactoryBean可以将声明式事务相关的关注点集中起来，一定程度上减少了配置的工作量。不过话又说回来了，如果应用程序中仅有少量的业务对象需要配置声明式事务，那么配置的工作量还算说的过去，一旦需要声明式事务的业务对象数量增加，采用这种近乎“手工作坊式”的配置方式就会“拖后腿”了。这时，我们自然会想到AOP中的自动代理机制，而下面正是针对如何使用自动代理对声明式事务进行管理的内容。

3. 使用BeanNameAutoProxyCreator

使用BeanNameAutoProxyCreator进行声明式事务管理进一步地简化了配置的工作。当所有的声明式事务相关装备一次到位之后，要为新的业务对象添加声明式事务支持，唯一要做的就是，在为该业务对象添加bean定义的时候，同时将它的beanName添加到BeanNameAutoProxyCreator管理的beanNames列表中。

使用BeanNameAutoProxyCreator为业务对象提供声明式事务支持，通常配置如代码清单20-14所示。

代码清单20-14 使用BeanNameAutoProxyCreator自动化声明式事务管理配置示例

```
<beans>
    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"➥
    destroy-method="close">
        <property name="url" value="jdbc:mysql://localhost/databaseName"/>
        <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
        <property name="username" value="..."/>
        <property name="password" value="..."/>
    </bean>

    <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <bean id="transactionManager" class="org.springframework.jdbc.datasource. ➥
    DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <bean id="transactionInterceptor" class="org.springframework.transaction.interceptor. ➥
```

```
TransactionInterceptor">
    <property name="transactionManager" ref="transactionManager"/>
    <property name="transactionAttributeSource">
        <value>
org.spring21.package.IQuoteService.getQuate*=PROPAGATION_SUPPORTS,readOnly,timeout_20
org.spring21.package.IQuoteService.saveQuote=PROPAGATION_REQUIRED
org.spring21.package.IQuoteService.updateQuote=PROPAGATION_REQUIRED
org.spring21.package.IQuoteService.deleteQuote=PROPAGATION_REQUIRED
        </value>
    </property>
</bean>

<bean id="autoProxyCreator" class="org.springframework.aop.framework.autoproxy. ➥
BeanNameAutoProxyCreator">
    <property name="interceptorNames">
        <list>
            <value>transactionInterceptor</value>
        </list>
    </property>
    <property name="beanNames">
        <list>
            <idref bean="quoteService"/>
            ...
        </list>
    </property>
</bean>

<bean id="quoteService" class="...QuoteService">
    <property name="jdbcTemplate" ref="jdbcTemplate"/>
</bean>

<bean id="client" class="...QuoteSerivceClient">
    <property name="quoteService" ref="quoteService"/>
</bean>
</beans>
```

现在，我们只需要正常地向IoC容器的配置文件中增加相应的业务对象bean定义。BeanNameAutoProxyCreator将根据TransactionInterceptor提供的事务管理功能，为添加到它的beanNames列表的所有业务对象自动添加事务支持（当然，本质上是为其生成动态代理对象）。

无论应用中业务对象数量多少，使用BeanNameAutoProxyCreator都可以很便捷地处理这些业务对象的声明式事务需求。不过，可能在实际的开发过程中，我们依然会感觉使用BeanNameAutoProxyCreator有其不够便捷之处。好消息就是，如果我们的应用程序可以，或者已经升级到Spring 2.*x*，那么，使用基于XSD的配置方式吧！

4. 使用Spring 2.*x*的声明事务配置方式

Spring 2.*x*后提供的基于XML Schema的配置方式，专门为事务管理提供了一个单独的命名空间用于简化配置，结合新的tx命名空间，现在的声明式事务管理看起来要清晰许多（见代码清单20-15）。

代码清单20-15　基于XSD的声明式事务配置代码示例

```
<beans xmlns="http://www.springframework.org/schema/beans" ➥
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ➥
xmlns:util="http://www.springframework.org/schema/util" ➥
xmlns:jee="http://www.springframework.org/schema/jee" ➥
xmlns:lang="http://www.springframework.org/schema/lang" ➥
xmlns:aop="http://www.springframework.org/schema/aop" ➥
xmlns:tx="http://www.springframework.org/schema/tx" ➥
xsi:schemaLocation="http://www.springframework.org/schema/beans ➥
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd ➥
```

```
    http://www.springframework.org/schema/aop ➥
    http://www.springframework.org/schema/aop/spring-aop-2.0.xsd ➥
    http://www.springframework.org/schema/tx ➥
    http://www.springframework.org/schema/tx/spring-tx-2.0.xsd">

    <aop:config>
        <aop:pointcut id="txServices" expression="execution(* cn.spring21. ➥
        unveilspring.IQuoteService.*(..))"/>
        <aop:advisor pointcut-ref="txServices" advice-ref="txAdvice"/>
    </aop:config>

    <tx:advice id="txAdvice" transaction-manager="transactionManager">
        <tx:attributes>
            <tx:method name="getQuate*" propagation="SUPPORTS" read-only="true" ➥
            timeout="20"/>
            <tx:method name="saveQuote"/>
            <tx:method name="updateQuote"/>
            <tx:method name="deleteQuote"/>
        </tx:attributes>
    </tx:advice>

    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" ➥
    destroy-method="close">
        <property name="url" value="jdbc:mysql://localhost/dbName"/>
        <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
        <property name="username" value="..."/>
        <property name="password" value="..."/>
    </bean>

    <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <bean id="transactionManager" class="org.springframework.jdbc.datasource. ➥
    DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <bean id="quoteService" class="...QuoteService">
        <property name="jdbcTemplate" ref="jdbcTemplate"/>
    </bean>

    <bean id="client" class="...QuoteSerivceClient">
        <property name="quoteService" ref="quoteService"/>
    </bean>
</beans>
```

<tx:advice>是专门为声明事务Advice而设置的配置元素，底层当然还是我们的Transaction-Interceptor，只是披一件新衣裳而已。<tx:advice>的transaction-manager指定了它要使用的事务管理器是哪一个。如果容器中事务管理器的beanName恰好就是transactionManager，那么可以不明确指定。<tx:advice>内部由<tx:attributes>提供声明式事务所需要的元数据映射信息，每条映射信息对应一个<tx:method/>元素声明。<tx:method/>只有name属性是必须指定的，其他的属性代表事务定义的其他内容，比如propagation用于指定传播行为，isolation用于指定隔离度，timeout用于指定事务的超时时间等。如果不明确指定的话，将采用DefaultTransactionDefinition的设置内容。表20-1是<tx:method/>可用的属性的详细列表①。

① 实际上，只要有一个支持XSD的XML编辑器，我们根本就不用记住每个属性的具体特征。通常，这种编辑器都会有提示功能，只要提示的每个属性你知道它具体代表什么意思，剩下的就是个选择的问题。

表20-1　<tx:method/>属性对照表

属 性 名	说　明
name	事务元数据将要加之于上的业务方法名称，可以使用*通配符
propagation	事务的传播行为，不明确指定的话，默认值为REQUIRED。该名称去掉了TransactionDefintion中常量名称的前缀，在指定的时候，需要注意
isolation	用于指定事务的隔离度。不明确指定的话，默认值采用DEFAULT，也是TransactionDefintion中常量名称去掉前缀
timeout	事务的超时时间，默认值为-1
read-only	指定事务是否为只读事务，默认值为false
rollback-for	用于指定能够触发事务回滚的异常类型，比如rollback-for="cn.spring21.unveilspring.QuoteException"。如果有多个异常类型需要指定，各类型之间可以通过逗号分隔
no-rollback-for	与rollback-for正好相反，即使抛出no-rollback-for指定的异常类型也不回滚事务

通过<tx:advice>指定的事务信息，需要有Spring AOP的支持才能织入到具体的业务对象，所以，剩下的工作实际上是AOP的配置了，如下所示：

```
<aop:config>
    <aop:pointcut id="txServices" expression="execution(* cn.spring21.unveilspring. ➥
    IQuoteService.*(..))"/>
    <aop:advisor pointcut-ref="txServices" advice-ref="txAdvice"/>
</aop:config>
```

在Spring AOP部分已经说过，<aop:config>底层也是依赖于自动代理机制。所以，我一直强调，新的基于XSD的配置方式，只是换了一身简洁明快的外衣，而我想让大家看到的，则是外衣里面的东西。

> **注意**　更多使用<tx:advice>的内容可以参照Spring 2.x之后的参考文档，其中有更多详细内容，比如配置多个<tx:advice>以区分不同的事务需求等内容。

20.2.3　注解元数据驱动的声明式事务

随着Java 5（Tiger）的发布，注解越来越受到开发人员的关注和喜爱，如果你的应用程序构建在Java 5或者更高版本的虚拟机上的话，那么恭喜你，现在你也可以使用Spring提供的基于注解的声明式事务管理了[①]。

注解元数据驱动的声明式事务管理的基本原理是，将对应业务方法的事务元数据，直接通过注解标注到业务方法或者业务方法所在的对象上，然后在业务方法执行期间，通过反射读取标注在该业务方法上的注解所包含的元数据信息，最终将根据读取的信息为业务方法构建事务管理的支持。

Spring定义了org.springframework.transaction.annotation.Transactional用于标注业务方法所对应的事务元数据信息。通过Transactional，可以指定与<tx:method/>几乎相同的信息。当然，现在不用指定方法名称了，因为Transactional直接标注到了业务方法或者业务方法所在的对象定义上。通过查看Transactional的定义（见代码清单20-16），我们可以获取所有可以指定的事务

---

① 当然，即使不使用Java 5，也不意味着不能使用源码中的元数据驱动的声明式事务，比如我们可以使用commons attributes（http://commons.apache.org/attributes/）。但是，Java 5的注解可以为我们带来更多的东西，比如编译器检查，IDE良好的重构支持等。如果当前应用无法使用Java 5，而又要使用源代码中元数据驱动的声明式事务，可以参照Spring参考文档中提供的使用commons attributes的相关信息。

定义内容。

代码清单20-16 Transactional的定义

```
@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
public @interface Transactional
{
    Propagation propagation() default Propagation.REQUIRED;
    Isolation isolation() default Isolation.DEFAULT;
    int timeout() default TransactionDefinition.TIMEOUT_DEFAULT;
    boolean readOnly() default false;
    Class<? extends Throwable>[] rollbackFor() default {};
    String[] rollbackForClassName() default {};
    Class<? extends Throwable>[] noRollbackFor() default {};
    String[] noRollbackForClassName() default {};
}
```


要为QuoteService添加基于注解的声明式事务管理，需要为其添加Transactional以标注必要的事务管理信息，如代码清单20-17所示。

代码清单20-17 使用@Transactional标注事务管理目标类的代码示例

```
@Transactional
public class QuoteService implements IQuoteService
{
    private JdbcTemplate jdbcTemplate;

    @Transactional(propagation=Propagation.SUPPORTS,readOnly=true,timeout=20)
    public Quote getQuate() {
        return (Quote)getJdbcTemplate().queryForObject("SELECT * FROM fx_quote where ➥
        quote_id=2", new RowMapper(){

            public Object mapRow(ResultSet rs, int row) throws SQLException {
                Quote quote = new Quote();
                // ...
                return quote;
            }});
    }
    @Transactional(propagation=Propagation.SUPPORTS,readOnly=true,timeout=20)
    public Quote getQuateByDateTime(DateTime dateTime) {
        throw new NotImplementedException();
    }
    public void saveQuote(Quote quote) {
        throw new NotImplementedException();
    }
    public void updateQuote(Quote quote) {
        throw new NotImplementedException();
    }
    public void deleteQuote(Quote quote) {
        throw new NotImplementedException();
    }
    public JdbcTemplate getJdbcTemplate() {
        return jdbcTemplate;
    }
    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }
```

}

如果将@Transactional标注为对象级别的话，对象中的方法将“继承”该对象级别上的@Transactional的事务管理元数据信息。如果某个方法有特殊的事务管理需求，可以在方法级别添加更加详细的@Transactional设定，比如getQuote*()方法。通过将相同的事务管理行为提取到对象级别的@Transactional，可以有效地减少标注的数量。如果不为@Transactional指定自定义的一些设定，它也会像<tx:method/>那样采用与DefaultTransactionDefinition一样的事务定义内容。

只通过@Transactional标注业务对象以及对象中的业务方法，并不会为业务方法带来任何事务管理的支持。@Transactional只是一个标志而已，需要我们在执行业务方法的时候，通过反射读取这些信息，并根据这些信息构建事务，才能使这些声明的事务行为生效。这就好像代码清单20-18中的代码所演示的那样。

代码清单20-18　@Transactional声明式事务实现原型代码示例

```
...
public Quote getQuate()
{
    try {
        Method method = quoteService.getClass().getDeclaredMethod("getQuate", null);
        boolean isTxAnnotationPresent = method.isAnnotationPresent(Transactional.class);
        if(!isTxAnnotationPresent)
        {
            return (Quote)quoteService.getQuate();
        }
        Transactional txInfo = method.getAnnotation(Transactional.class);
        TransactionTemplate transactionTemplate = new TransactionTemplate ➥
        (transactionManager);
        if(!txInfo.propagation().equals(Propagation.REQUIRED))
            transactionTemplate.setPropagationBehavior(txInfo.propagation().value());
        if(txInfo.readOnly())
            transactionTemplate.setReadOnly(true);
        // ...
        return (Quote)transactionTemplate.execute(new TransactionCallback(){
            public Object doInTransaction(TransactionStatus txStatus) {
                return quoteService.getQuate();
            }});
    } catch (SecurityException e) {
        e.printStackTrace(); // 不要这样做
        return null;
    } catch (NoSuchMethodException e) {
        e.printStackTrace(); // 不要这样做
        return null;
    }
}
...
```

不过，我们不用自己去写这些底层的逻辑了，通过在容器的配置文件中指定如下一行配置，这些搜寻注解、读取内容、构建事务等工作全都由Spring的IoC容器搞定：

```
<tx:annotation-driven transaction-manager="transactionManager"/>
```

所以，使用注解元数据驱动的声明式事务管理，基本上就需要做两件事：

- 使用@Transactional标注相应的业务对象以及相关业务方法。这一步通常由业务对象的开发者统一负责，@Transactional的使用使得元数据以及业务逻辑的实现全部集中到了一处，有了IDE的支持，管理起来更是得心应手。

- 在容器的配置文件中设定事务基础设施。我们需要添加<tx:annotation-driven transaction-manager="transactionManager"/>以便有人使用我们所标注的@Transactional，并且，需要给出相应的事务管理器，要进行事务管理，没有它可不行。

对于QuoteService来说，在已经使用@Transactional标注了相应的事务管理信息之后，剩下的就是对应容器配置的内容了，详情见代码清单20-19。

代码清单20-19 使用@Transactional进行声明式事务管理的完整配置代码示例

```
<tx:annotation-driven transaction-manager="transactionManager"/>

<bean id="transactionManager" class="org.springframework.jdbc.datasource. ➥
DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" ➥
destroy-method="close">
    <property name="url" value="jdbc:mysql://localhost/dbName"/>
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="username" value="..."/>
    <property name="password" value="..."/>
</bean>

<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
    <property name="dataSource" ref="dataSource"/>
</bean>

<bean id="quoteService" class="...QuoteService">
    <property name="jdbcTemplate" ref="jdbcTemplate"/>
</bean>

<bean id="client" class="...QuoteSerivceClient">
    <property name="quoteService" ref="quoteService"/>
</bean>
```

至此，基于注解的声明式事务就算大功告成了。

> **提示** Spring推荐将@Transactional标注于具体的业务实现类或者实现类的业务方法上。之所以如此，是因为Spring AOP可以采用两种方式来生成代理对象（动态代理或者CGLIB）。如果将@Transactional标注于业务接口的定义上，那么，当使用动态代理机制构建代理对象时，读取接口定义上的@Transactional信息是没有问题的。可是当使用CGLIB构建代理对象的时候，则无法读取接口上定义的@Transactional数据。

20.3 小结

本章主要阐述了如何使用Spring事务抽象框架进行事务管理的相关内容，分别就如何使用Spring进行编程式事务管理和声明式事务管理进行了详细的阐述。学习完本章，相信大家已经可以游刃于事务管理相关的日常开发工作了。

Spring的事务抽象框架中使用到一些日常开发过程中常用的理念和方法。接下来，我们将尝试挖掘一下这些宝贵的经验，以为我用。

第21章 Spring事务管理之扩展篇

本章内容

- 理解并活用ThreadLocal
- 谈Strategy模式在开发过程中的应用
- Spring与JTA背后的奥秘

21.1 理解并活用 ThreadLocal

之前在事务管理器实现原型的讲解中，我们展示了ThreadLocal在Spring的事务管理框架中所起的核心作用，即通过ThreadLocal的使用来避免connection-passing方式最初的尴尬局面。不过，我们依然认为，这只是反映了ThreadLocal使用中的一个侧面。为了让大家更加深入地理解ThreadLocal，并在之后的开发活动中灵活使用它，特增加了本章内容。

21.1.1 理解 ThreadLocal 的存在背景

ThreadLocal是Java语言提供的用于支持线程局部变量（thread-local variable）的标准实现类。我们可以在称为Thread-Specific Storage Pattern[①]的多线程编程模式中一窥ThreadLocal的典型应用场景。

国内的Java技术社区对ThreadLocal的理解可以说是百家争鸣，异常热闹。有的将ThreadLocal和synchronized放在一起讨论，有的说它们之间没有任何关系，还有的说ThreadLocal不是用来解决多线程环境下的对象共享问题的，这些观点通常会令许多开发人员对ThreadLocal更加迷惑。不能说这些观点是不对的，但我认为这些都没有说到点子上！那么，ThreadLocal到底是为何而生呢？它又是否与之前的观点有什么千丝万缕的联系呢？下面让我带大家一起来探索这些问题的答案。

单单从程序层面来看，我们所编写的代码实际上是在管理系统中各个对象的相关状态。如果不能够对各个对象状态的访问进行合理的管理，对象的状态将被破坏，进而导致系统的不正常运行。特别是多线程环境下，多个线程可能同时对系统中的单一或者多个对象状态进行访问，如果不能保证在此期间的线程安全，将会把整个系统带向崩溃的边缘。

为了保证整个应用程序的线程安全，我们可以采用多种方式。不过，在此之前，我们不妨先来看看打架与我们管理线程安全问题有何相似之处。现在你就是系统中的一个对象，假设某一天你出门在外遇到三名歹徒（当然，仅仅是假设，不用害怕，呵呵），你毫无退路，只能一搏，你会采用什么样的策略来保证你的人身安全呢？毕竟，我们最终的目的都是保证你这个对象的状态始终处于“正确”的状态，对吧？

---

① 在《Java多线程设计模式》一书中，对Thread-Specific Storage Pattern有详细的介绍和讲解。

策略一，你是个练家子，而且身边并无他人相助，所以，你只能同时对付三名歹徒。为了不受伤害，你是辗转腾挪，左躲右闪，尽量保证每一时刻都只对付一名歹徒，以便最终能够全身而退。如果把三名歹徒对你的攻击顺序比作三个线程的话，你实际上是在用同步（Synchronization）的方式来管理这三个线程对你的访问。

使用同步方式来管理多个线程对对象状态的访问以保证应用程序的线程安全是最常用的方式。不过，你也看到了，你需要很辛苦地对待，稍一不慎，就有可能受伤。所以，这时我们就在想，我要是孙悟空有多好。

"但见那孙猴子揪一撮猴毛一吹，片刻化作多个分身"，"小的们，给我上"。现在，我们再也不用苦熬了，让一个分身对付一个歹徒（小妖），让他们在各自的线程内各自折腾去吧！不用一对三，当然也就不可能破坏到我的完好状态咯！

这就是策略二，通过避免对象的共享，同样达到线程安全的目的。你想啊，都在各自的线程内跟各自的分身折腾去了，自然也就不会需要同步对单一共享资源的访问了。ThreadLocal的出现，实际上就是帮助我们以策略二的方式来管理程序的线程安全。只要当前环境允许，能不共享的尽量不共享，反而更容易管理应用程序的线程安全。

综上来说，同步和ThreadLocal在横向上可能没有任何的关系，但从纵向上看，它们实际上都服务于同一个目的，那就是帮助我们实现应用程序的线程安全。另外，说ThreadLocal不是用来解决多线程环境下对象共享问题的，也就更好解释了。ThreadLocal的目的是通过避免对象的共享来保证应用程序实现中的线程安全。共享对象是ThreadLocal尽量避免的。如果要管理的对象非要共享，ThreadLocal自然不会理会这码子事儿了。

21.1.2 理解 ThreadLocal 的实现

我们已经了解了ThreadLocal因何而生，现在该是我们探索ThreadLocal又是如何运作的时候了，它到底是如何来完成它的职责的呢？

虽然是通过ThreadLocal来设置特定于各个线程的数据资源，但ThreadLocal自身不会保存这些特定的数据资源。因为数据资源特定于线程的，自然是由每个线程自己来管理了。每个Thread类都有一个ThreadLocal.ThreadLocalMap类型的名为threadLocals的实例变量，它就是保持那些通过ThreadLocal设置给这个线程的数据资源的地方。当通过ThreadLocal的set(data)方法来设置数据的时候，ThreadLocal会首先获取当前线程的引用，然后通过该引用获取当前线程持有的threadLocals，最后，以当前ThreadLocal作为Key，将要设置的数据设置到当前线程，如下所示：

```
Thread thread = Thread.currentThread();
ThreadLocalMap threadlocalmap = thread.threadLocals;
...
threadlocalmap.set(this, obj);
```

而至于余下的get()之类的方法，基本上也是同样的道理，都是首先取得当前线程，然后根据每个方法的语义，对当前线程所持有的threadLocals中的数据进行操作。

实际上，ThreadLocal就好像是一个窗口，通过这个窗口，我们可以将特定于线程的数据资源绑定到当前线程，也可以通过这个窗口获取绑定的数据资源，当然，更可以解除之前绑定到当前线程的数据资源。在整个线程的生命周期内，我们都可以通过ThreadLocal这个窗口与当前线程打交道。

为了更好地理解Thread与ThreadLocal之间的关系，我们不妨设想一下城市的公交系统（见图21-1）。

城市中的各条公交线路就好像我们系统中的那一个个线程，在各条公交线路上，会有相应的公交

车辆，这些公交车辆就好像Thread的threadLocals，用来运送特定于该条线路的乘客（数据资源）。为了乘客可以乘车或者下车，各条公交线路沿线都会设置多个乘车点（Bus Stop），而这些乘车点实际上就是ThreadLocal。虽然同一个乘车点可能会有多条公交线路共用，但同一时间，乘客只会搭乘他要乘坐并且当前经过的公交车。这与ThreadLocal和Thread的关系是类似的，虽然同一个ThreadLocal可以为多个线程指定数据资源，但只会将数据资源指定到当前的线程。

至此，你应该再也不会感觉ThreadLocal神秘了吧？

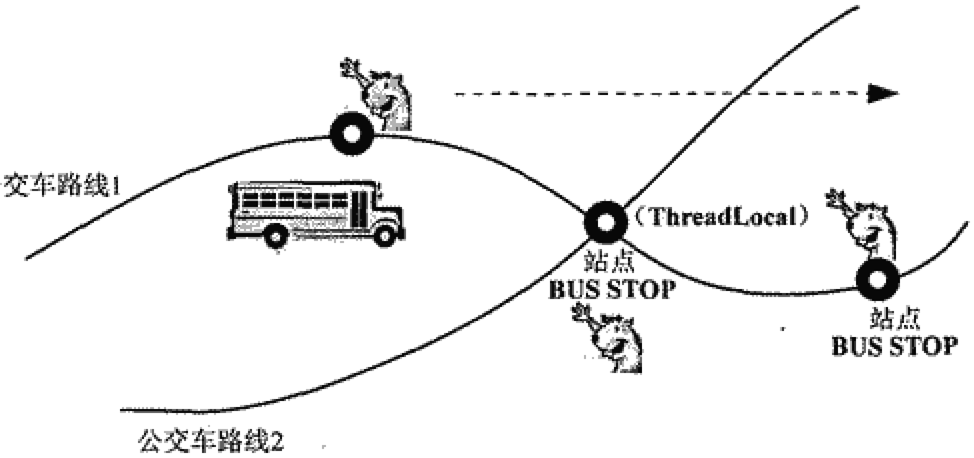

图21-1　公交线路和车站与线程和ThreadLocal之间的类比示意图

21.1.3　ThreadLocal 的应用场景

ThreadLocal的概念和提供的功能很简单，但如果能够充分发挥ThreadLocal的能力，将会为我们的开发工作带来意想不到的效果。

基本上，我们可以从两个方面来看待并灵活应用ThreadLocal（见图21-2）。

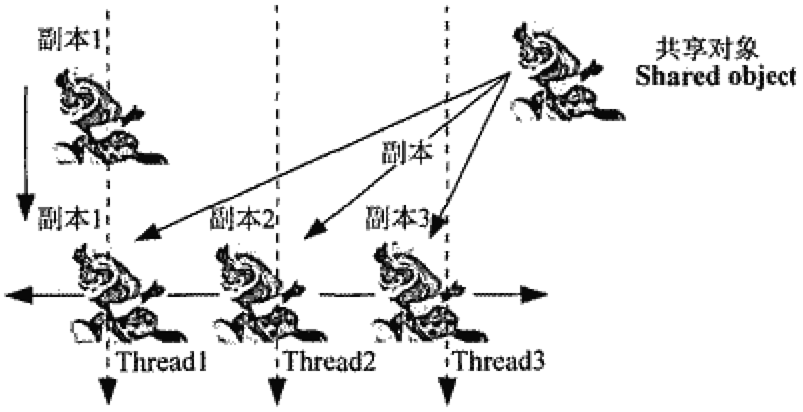

图21-2　ThreadLocal功能分析图

- 横向上看，我们是更注重于ThreadLocal横跨多个线程的能力，这当然是ThreadLocal最初的目的所在。为了以更加简单地方式来管理应用程序的线程安全，ThreadLocal干脆将没有必要共享的对象不共享，直接为每个线程分配一份各自特定的数据资源。
- 纵向上看，我们则着眼于ThreadLocal在单一线程内可以发挥的能力，通过ThreadLocal设置的特定于各个线程的数据资源，可以随着所在线程的执行流程“随波逐流”。

当然，两个方面不是严格独立的，更多时候它们则是相互依存、紧密结合的。

在充分发挥ThreadLocal两方面能力的基础上，我们可以发掘出以下几种ThreadLocal的应用场景。

- **管理应用程序实现中的线程安全。**对于某些有状态的或者非线程安全的对象，我们可以在多线程程序中为每个线程都分配相应的副本，而不是让多个线程共享该类型的某个对象，从而避免了需要协调多个线程对这些对象进行访问的“危险”工作。

  在使用JDBC进行数据访问的过程中，Connection对象就属于那种有状态并且非线程安全的

类。所以，为了保证多个线程使用Connection进行数据访问过程中的安全，我们通过ThreadLocal为每个线程分配了一个它们各自持有的Connection，从而避免了对单一Connection资源的争用。毕竟，在JDBC中是一个Connection对应一个事务。如果所有的线程都共用一个Connection的话，那么整个事务管理就有点儿失控的感觉了。

在之前部分，当以简化的形式向读者展示Spring的事务框架是如何对Connection进行管理的时候，我们只是强调了使用ThreadLocal来避免“connection passing”的尴尬，而实际上，通过ThreadLocal来保证Connection在多线程环境下的正确使用，应该也是Spring的事务框架使用ThreadLocal进行Connection管理的原因之一。

- **实现当前程序执行流程内的数据传递。** 这种场景下，我们更多关注的是在单一的线程环境中使用ThreadLocal。在剥离线程安全等因素的考虑之后，“connection passing”实际上就可以看作这个场景下的一例。

  除此之外，我们也可以通过ThreadLocal来跟踪保存在线程内的日志序列，在程序执行的任何必要执行点将系统跟踪信息添加到ThreadLocal，然后在合适的时点取出分析。我们还可以通过ThreadLocal来保存某种全局变量，在线程内执行流程的某个时点设置该全局变量，然后在合适的位置获取其值以做某些判断工作等。只要你愿意，任何合适的数据都可以通过“ThreadLocal在单一线程内传递数据”这一功能进行传递。

  采用ThreadLocal进行当前执行流程内的数据传递，可以避免耦合性很强的方法参数形式的数据传递方式。但这有些像是让数据随着“暗流”漂泊的意思，一旦处理不当就会出现“触礁”之类的事故，比如资源没有适当的清理导致系统行为差异。所以，通常应该通过一组框架类来规范并屏蔽对ThreadLocal的直接操作，尽量避免应用代码的直接接触。

- **某些情况下的性能优化。** 有些情况下，系统中一些没有必要共享的对象被设置成了共享，为了保证应用程序的线程安全以及对象状态的正确，我们往往就得通过同步等方式对多线程的访问进行管理和控制。这时，各个线程在走到这个共享对象的时候就得排队，一个一个地对该共享对象进行访问。显然，这将严重影响系统的处理性能，比如去银行取款的时候，就一个窗口在营业，而你又急着取钱，可以想象一下你现在是一种什么样的心情。所以，能够避免共享的时候，就尽量不要共享，多开几个营业窗口，要比单一营业窗口的处理速度快得多。

  某些情况下，通过ThreadLocal这种“以空间换时间”的方式来管理对象访问，可以收到更好的响应效果。

- **per-thread Singleton。** 当某项资源的初始化代价有些大，并且在整个执行流程中还会多次访问它的时候，为了避免在访问时每次都需要去初始化该项资源，我们可以在第一次将该资源初始化完成之后，直接通过ThreadLocal将其绑定到当前线程，之后，所有对该资源的访问都从当前线程获取即可。

这实际上与“实现当前程序执行流程内的数据传递”的应用场景很相似。不过，该场景更侧重于资源管理，所以，单独罗列在此也不为过了。

我想，灵活运用ThreadLocal可以在更多场景中发挥作用，故希望大家也能够在平时的开发过程中发掘出更多ThreadLocal的应用场景。

21.1.4 使用 ThreadLocal 管理多数据源切换的条件

第16章，介绍了如何实现一个简单的AbstractRountingDataSource原型来管理多个数据源之间的切换功能。当时的原型实现可能让你觉得过于简单而不甚真实。这里我们引入ThreadLocal来协助

管理多数据源切换的条件，以期抛砖引玉，使得你在日常的开发工作中灵活运用ThreadLocal带给我们的便利。

在多数据源的切换过程中，切换的条件可能随着应用程序的需求而各异，而且，通常不会像我们的AbstractRoutingDataSource原型实现那样，只需要内部条件就可以实现数据源切换的判断。更多时候，需要外部条件的介入，这就会有一个问题，如何为AbstractRoutingDataSource的实现子类传入这些外部条件相关的数据？ThreadLocal这个时候就可以派上用场。

我们的思路是，通过ThreadLocal保存每个数据源所对应的标志（该标志我们以枚举类的形式给出），AbstractRoutingDataSource在通过determineCurrentLookupKey()获取对应数据源的键值的时候，直接从ThreadLocal获取当前线程所持有的数据源对应标志然后返回。而至于说什么情况下使用哪个具体的数据源的问题，则是由应用程序的需求来决定，只要在必要的地方，将所要使用的具体数据源的对应标志通过ThreadLocal绑定到当前线程即可。

使用ThreadLocal管理多数据源切换条件的AbstractRoutingDataSource实现流程如下。

(1) 假设我们有MAIN、INFO和DBLINK三个数据源可用，第一步要做的事情就是先给出一个枚举类，其中定义了对应这三个数据源的标志，如下所示：

```
public enum DataSources {
    MAIN,INFO,DBLINK;
}
```

(2) 我们定义所使用的ThreadLocal（见代码清单21-1），没有“车站”，我们可没法上车啊！

代码清单21-1　持有ThreadLocal的DataSourceTypeManager类定义

```
public class DataSourceTypeManager
{
    private static final ThreadLocal<DataSources> dsTypes = new ThreadLocal<DataSources>(){
        @Override
        protected DataSources initialValue() {
            return DataSources.MAIN;
        }
    };

    public static DataSources get()
    {
        return dsTypes.get();
    }

    public static void set(DataSources dataSourceType)
    {
        dsTypes.set(dataSourceType);
    }

    public static void reset()
    {
        dsTypes.set(DataSources.MAIN);
    }
}
```

(3) 有了标志枚举类和相应的ThreadLocal定义之后，就可以实现我们的AbstractRoutingDataSource了，如下代码所示：

```
public class ThreadLocalVariableRountingDataSource extends AbstractRoutingDataSource {
```

```
    @Override
    protected Object determineCurrentLookupKey() {
        return DataSourceTypeManager.get();
    }
}
```

(4) 现在我们需要将ThreadLocalVariableRountingDataSource以及相关的依赖注册到IoC容器中，如代码清单21-2所示。

代码清单21-2 基于ThreadLocal的多数据源切换相关类配置示例

```
<bean id="mainDataSource" class="org.apache.commons.dbcp.BasicDataSource" ➥
destroy-method="close">
    <property name="url" value=".."/>
    <property name="driverClassName" value=".."/>
    <property name="username" value=".."/>
    <property name="password" value="."/>
</bean>

<bean id="infoDataSource" class="org.apache.commons.dbcp.BasicDataSource" ➥
destroy-method="close">
    <property name="url" value=".."/>
    <property name="driverClassName" value=".."/>
    <property name="username" value=".."/>
    <property name="password" value="."/>
</bean>

<bean id="dblinkDataSource" class="org.apache.commons.dbcp.BasicDataSource" ➥
destroy-method="close">
    <property name="url" value=".."/>
    <property name="driverClassName" value=".."/>
    <property name="username" value=".."/>
    <property name="password" value="."/>
</bean>

<bean id="dataSource" class="..ThreadLocalVariableRountingDataSource">
    <property name="defaultTargetDataSource" ref=""/>
    <property name="targetDataSources">
        <map key-type="..DataSources">
            <entry key="MAIN" value-ref="mainDataSource"/>
            <entry key="INFO" value-ref="infoDataSource"/>
            <entry key="DBLINK" value-ref="dblinkDataSource"/>
        </map>
    </property>
</bean>
```

注意，我们在配置ThreadLocalVariableRountingDataSource所使用的多个目标数据源的时候，使用了<map>的key-type属性指明了键值的类型，否则，就得通过其他的方式来确定枚举类的各值作为key与目标数据源之间的对应关系。

(5) 万事俱备之后，我们就可以使用如下代码使得数据源的切换生效：

```
DataSourceTypeManager.set(DataSources.INFO);
// 或者
DataSourceTypeManager.set(DataSources.DBLINK);
...
```

此后通过ThreadLocalVariableRountingDataSource所进行的数据访问，则会使用我们所设定的具体数据源。

至于说以上代码要放在什么地方，可以因应用程序而异，比如我们可以在程序或者某个线程启动之后进行设置，也可以在某个AOP的拦截器中，根据条件判断来进行设定以实现数据源的切换，等等。

至此，打完收功。怎么样？大家现在是否对ThreadLocal的应用跃跃欲试了呢？

21.2　谈 Strategy 模式在开发过程中的应用

Strategy模式的本意是封装一系列可以互相替换的算法逻辑，使得具体算法的演化独立于使用它们的客户端代码。为了理解为什么要这么做，我们不妨来看如下这个具体的场景。

> 在一个信贷系统中，通常会提供多种还款方式，比如等额本金还款方式、等额本息还款方式、一次还本付息方式等，那么，针对每位顾客所选择的还款方式，我们就需要按照这些还款方式的具体逻辑，为顾客计算每次所需要归还的本金以及利息的额度。如果要我们来实现这个根据还款方式计算额度的逻辑，我们会怎么做呢？

对于谙熟结构化编程或者面向对象编程不甚娴熟的开发人员来说，他们可能会直接使用多重条件语句来实现这段计算逻辑，如代码清单21-3所示。

代码清单21-3　使用多重条件语句实现多种算法逻辑的代码示例

```
public RepaymentDetails calculateRepayment(BigDecimal totalAmount,String customerId)
{
    RepaymentDetails details = new RepaymentDetails();
    Object type =  getRepaymentTypeByCustomerId(customerId);
    if(isEqualInterestRepaymentType(type))
    {
        BigDecimal interest = getEqualInterestOfCentrelBank();
        YearMonthDay repaymentInterval = getRepaymentIntervalByCustomerId(customerId);
        // 根据totalAmout和其他数据执行计算
    }
    if(isEqualPrincipalRepaymentType(type))
    {
        BigDecimal interest = getStandardInterestOfCentrelBank();
        YearMonthDay repaymentInterval = getRepaymentIntervalByCustomerId(customerId);
        // 根据totalAmout和其他数据执行计算
    }
    if(isOnceForAll(type))
    {
        BigDecimal interest = getStandardInterestOfCentrelBank();
        // 根据totalAmout和其他数据执行计算
    }
    ...

    return details;
}
```

当然，我们可以对这些代码做进一步的改进，但是，如果总体结构上不做任何变更的话，这种实现方式的问题会依然存在：

- 客户端代码与算法逻辑代码相互混杂，导致客户端代码过于复杂并且后期难以维护；
- 混杂的算法逻辑代码与客户端代码耦合性太强，算法的变更或者添加新的算法都会直接导致客户端代码的调整，使得客户端代码和算法逻辑代码无法独立演化；
- 几乎同一逻辑单元内实现的各种算法，无可避免地需要多重的条件语句来区分针对不同算法

所使用的数据或者对应算法的特定逻辑实现。

所以，该是Strategy模式登场的时间啦！

使用Strategy模式来重构这段代码的话，我们首先通过`RepaymentStrategy`定义来抽象还款逻辑算法，然后，针对不同的还款方式，给出`RepaymentStrategy`定义的不同实现。对于使用还款逻辑的客户端代码来说，它只需要获取相应的`RepaymentStrategy`引用，并调用接口公开的计算接口即可。整个图景如图21-3所示。

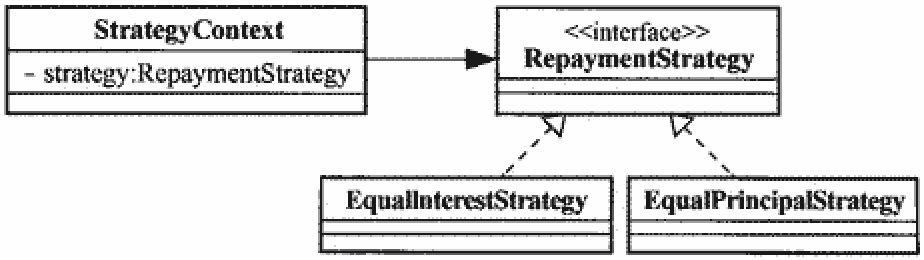

图21-3 `RepaymentStrategy`场景图

客户端代码只需要跟策略接口打交道，而算法的变更以及添加，对于使用策略接口进行计算操作的客户端代码来说，几乎没有任何影响。

虽然Strategy模式定义上强调的是对算法的封装，但我们不应该只着眼“算法”一词。实际上，只要能够有效地剥离客户端代码与特定关注点之间的依赖关系，Strategy模式就应该进入考虑之列。在这一点上，Spring框架的事务抽象就是一个很好的范例。通过将使用不同事务管理API进行事务管理的界定行为进行统一的抽象，客户端代码可以以透明的方式使用`PlatformTransactionManager`这一策略接口进行事务界定，即使具体的事务策略需要变更，对于客户端代码来说也不会造成过大的冲击。

Spring框架中使用Strategy模式的地方很多，除了本部分的事务抽象框架，还包括以下几处。

- 在IoC容器根据bean定义的内容，实例化相应bean对象的时候，会根据情况决定使用反射还是使用CGLIB来实例化相应的对象。`InstantiationStrategy`是容器使用的实例化策略的抽象接口，Spring框架默认提供了`CglibSubclassingInstantiationStrategy`和`SimpleInstantiationStrategy`两个具体实现类。
- Spring的Validation框架中，`org.springframework.validation.Validator`定义也是一个策略接口，具体的实现类将根据具体场景提供不同的验证逻辑。而这些具体验证逻辑的差异性，对于使用`Validator`进行数据验证的客户端代码来说，则是透明的。

除了在Spring框架内会发现Strategy模式的大量使用，我们也可以在其他的框架设计中发现Strategy模式的影子，比如最常用的jakarta commons logging中，`Log`接口就是一个策略接口，`Jdk14Logger`、`Log4JLogger`以及`SimpleLog`等都是具体的策略实现类。可见，只要针对同一件事情有多种选择的时候，我们都可以考虑用Strategy模式来统一一下抽象接口，为客户端代码“造福”。

Strategy模式的重点在于通过统一的抽象，向客户端屏蔽其所依赖的具体行为，但该模式并没有关注客户端代码应该如何来使用这些行为。一般来讲，客户端代码使用Strategy模式的方式可以简单划分为如下两种。

- **客户端整个生命周期内只依赖于单一的策略。**Spring提供的事务抽象可以归属这一类情况。使用`PlatformTransactionManager`进行事务界定的客户端代码，在其整个生命周期内只依赖于一个`PlatformTransactionManager`的实现类，或者`DataSourceTransactionManager`的实现类，或者`HibernateTransactionManager`的实现类等，这样的情况比较容易处理，直接为客户端代码注入所需要的策略实现类即可。
- **客户端整个生命周期内可能动态依赖多个策略。**比如我们的还款场景中，客户端可能需要根据每个顾客选择的还款方式，来决定使用哪个策略实现类为其计算对应的还款明细。对于这

种情况，你会发现，Strategy模式通常宣称的可以避免多重条件语句的问题，其实只是将其转移给了客户端代码而已（见代码清单21-4）。

代码清单21-4 常见的多策略接口动态调用代码示例

```
RepaymentStrategy strategy = fallbackStrategy();
...
public RepaymentDetails calculateRepayment(BigDecimal totalAmount,String customerId)
{
    Object type =  getRepaymentTypeByCustomerId(customerId);
    if(isEqualInterestRepaymentType(type))
    {
        strategy = EqualInterestStrategy();
    }
    if(isEqualPrincipalRepaymentType(type))
    {
        strategy = EqualPrincipalStrategy();
    }
    if(isOnceForAll(type))
    {
        strategy = OnceForAllStrategy();
    }
    ...
    return strategy.performCalculation();
}
```

不过，如果我们想真正地避免多重条件语句的话，也不是没有办法。最简单的方法就是提前准备一个具体策略类型与其对应条件之间的关系映射。对于还款的场景来说，我们可以这么做。

(1) 在客户端代码中声明一个对关系映射的依赖，如以下代码所示：

```
public class StrategyContext{
    private Map<Object,RepaymentStrategy> strategyMapping;

    ...
    // setter和getter方法
}
```

(2) 通过IoC容器注入所有客户端可能动态依赖的策略实现类实例（见代码清单21-5）。

代码清单21-5 策略关系映射配置示例

```
<bean id="strategyContext" class="...StrategyContext">
    <property name="strategyMapping">
        <ref local="strategyMapping"/>
    </property>
</bean>

<util:map id="strategyMapping">
    <entry key="EQAUL_INTEREST">
        <bean class="...EqualInterestStrategy"></bean>
    </entry>
    <entry key="EQAUL_PRINCIPAL">
        <bean class="...EqualPrincipalStrategy"></bean>
    </entry>
    <entry key="ONCE_FOR_ALL">
        <bean class="...OnceForAllStrategy"></bean>
    </entry>
</util:map>
```

(3) 在计算还款明细的方法中，使用还款策略的代码直接从关系映射中获取具体的策略即可，如代码清单21-6所示。

代码清单21-6 根据关系映射获取具体策略的代码示例

```
RepaymentStrategy strategy = fallbackStrategy();
...
public RepaymentDetails calculateRepayment(BigDecimal totalAmount,String customerId)
{
    Object type =  getRepaymentTypeByCustomerId(customerId);
    RepaymentStrategy strategy = strategyMapping.get(type);
    // 检查约束条件
    if(strategy == null)
        stargety = fallbackStrategy();
    return strategy.performCalculation();
}
```


> **提示** 除了使用IoC容器注入映射关系，我们还可以将对应应用程序的映射关系放到数据库或者其他外部配置文件，甚至注解中。通过IoC容器一次注入多个策略实例，可能需要占用多一些的系统资源。对于资源紧要的应用来说，可以考虑通过反射等方式按需构建具体策略实例，这个就留给你来完成吧！

在系统中合理的使用Strategy模式可以使得系统向着“高内聚，低耦合”的理想方向迈进，在改善应用程序代码结构的同时，进一步的提高产品质量。实际上，Strategy模式更是多态（Polymorphism）的完美体现。当你的OO内功修炼到“炉火纯青”的时候，也就发现，所谓的Strategy模式的概念，在你的脑海中或许已经淡然了。

21.3 Spring与JTA背后的奥秘

无论是Spring的参考文档，还是大多数介绍Spring的书籍，在提到使用Spring的JtaTransactionManager进行分布式事务管理的时候，都强调需要使用从应用服务器的JNDI服务获取的dataSource，而不是本地配置的普通dataSource（见图21-4）。

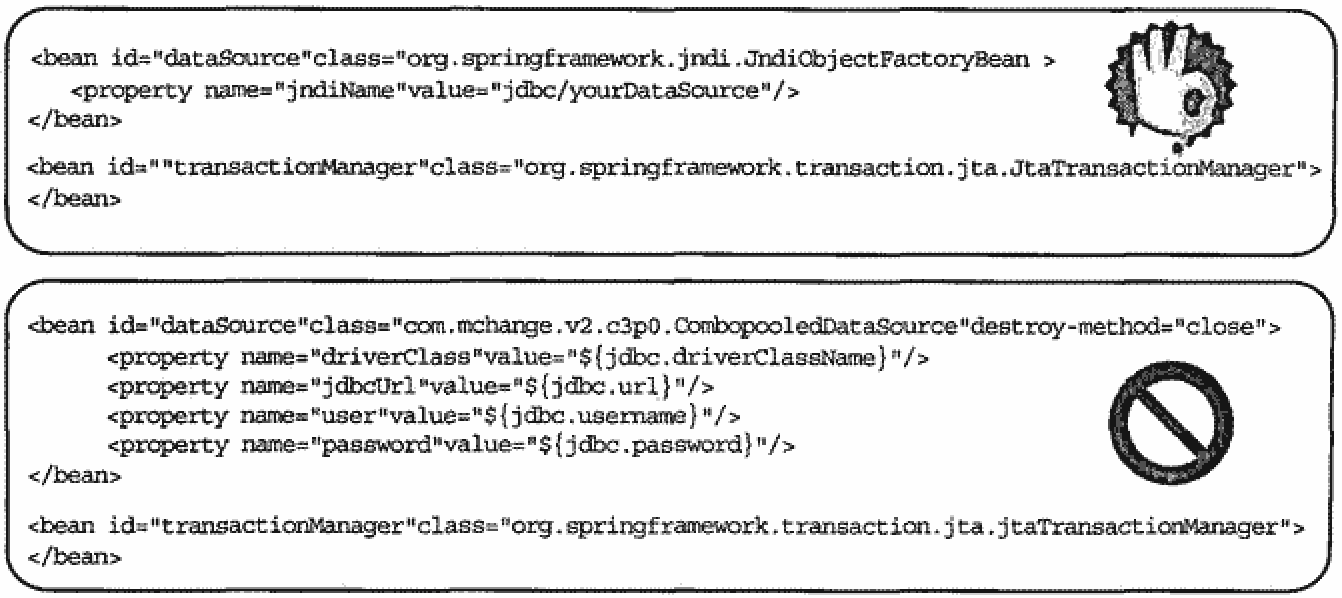

图21-4 JtaTransactionManager和DataSource之间的是与非

那么原因是什么呢？

我们知道，事务管理是要加之于具体的事务资源上的，所以，通常的PlatformTransactionManager的实现都会有相对应的事务资源的引用，比如，DataSourceTransactionManager需要指定DataSource，HibernateTransactionManager需要指定SessionFactory等。代码清单21-7再次演示了使用DataSourceTransactionManager的情况。

代码清单21-7　使用DataSourceTransactionManager的事务管理相关配置示例

```
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource" ➥
destroy-method="close">
    <property name="driverClass" value="${jdbc.driverClassName}"/>
    <property name="jdbcUrl" value="${jdbc.url}"/>
    <property name="user" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>

<bean id="transactionManager" class="org.springframework.jdbc.datasource. ➥
DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

从常识上来讲，像DataSourceTransactionManager这样才是正常的情况。可是，我们也看到，JtaTransactionManager确实没有明确指定依赖于哪个资源，却依然能够将需要加入分布式事务的资源纳入其管理范围。它是怎么做到的呢？如果揭开这个谜团，是否就能搞清楚在使用JtaTransactionManager的时候，要求我们使用从应用服务器的JNDI服务查找到的DataSource的真正原因呢？

在介绍JtaTransactionManager的时候，我们说到，JtaTransactionManager只是对各种具体的JTA实现产品提供的分布式事务管理功能进行了封装，JtaTransactionManager会将最终的工作委派给具体的JTA实现来做。所以，追根溯源，我们不得不追到JTA规范以及JTA实现中。

略去JTA以及X/Open规范千言不提，我们还是长话短说，直接进入正题吧！①

首先，具体的事务资源，RDBMS、MessageQueue等，要加入JTA管理的分布式事务，JTA规范要求其实现javax.transaction.xa.XAResource接口。所以，希望加入JTA管理的分布式事务的RM（Resource Manager，资源管理器）通常会提供相应的适配器（Adaptor），用于提供基于XAResource的分布式事务交互能力，比如关系数据库提供的支持XA的JDBC驱动程序，就是这样的适配器。这样，所有资源管理与事务交互的工作，基本上就由RM的适配器来统一管理了。

在想要参与JTA分布式事务的事务资源拥有了XAResource支持之后，JTA的javax.transaction.TransactionManager（我们称其为JTA TransactionManager，区别于Spring的JtaTransactionManager）与RM之间就可以进行通信，如图21-5所示。

适配器通常都有应答能力，这样，在JTA TransactionManager使用两阶段提交协议管理分布式事务的过程中，可以同每个RM进行交互。

不过，JTA TransactionManager在事务管理过程中要与哪些RM打交道，却不是由它自己说了算的。想要参与JTA分布式事务的RM何时何地甚至怎样加入JTA TransactionManager管理的分布式事务，也不是每个RM自己说了算。JTA TransactionManager与各个RM之间的联系要由ApplicationServer（一般意义上的TP Monitor）来进行协调！ApplicationServer为基于JTA的分布式事务提供运行时环境，并负责协调JTA TransactionManager与各RM之间的交互，整个过程类似于如下所叙述的。

① 实际上JTA规范本身不到100页，想要更深入地了解JTA的内容的话，看一下也无妨啊！

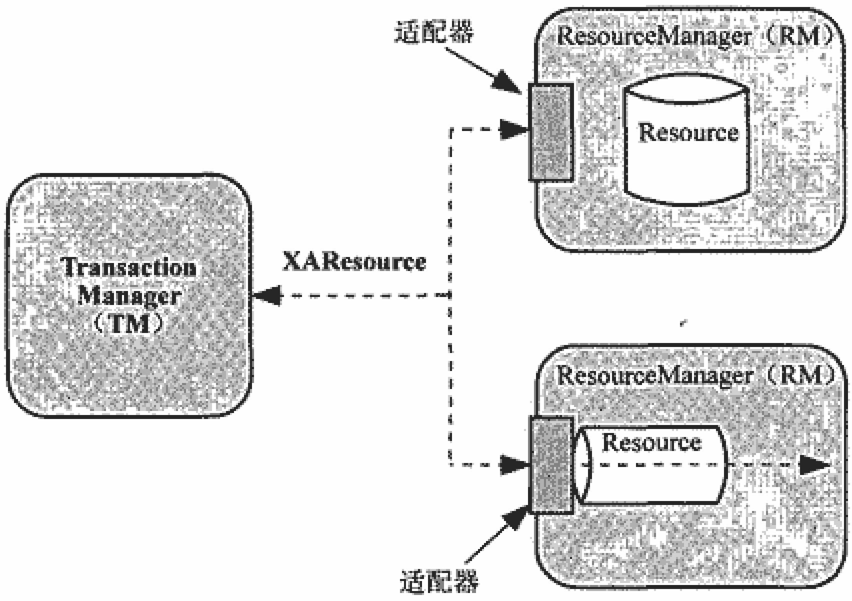

图21-5 TM与RM之间的通信

(1) ApplicationServer一开始当然要先通过JNDI绑定它的JTA实现中的UserTransaction或者TransactionManager具体实现类，这样，客户端应用程序就可以通过JNDI获取它们。现在客户端应用程序想要开始一个分布式事务，进而UserTransaction或者TransactionManager的相应方法被调用。

(2) ApplicationServer内部会要求TransactionManager为当前事务分配一个唯一的标志（以Xid表示），然后开始事务，并将当前事务绑定到当前线程。

(3) 客户端跟ApplicationServer要求相应的数据资源进行数据访问，ApplicationServer会跟RM的适配器要一个事务资源对象，我们暂且称之为TransactionalResource。该资源对象包含两部分，一部分是JTA TransactionManager需要与之交互的XAResource，另一部分是要公开给客户端应用程序使用的Connection资源，取得TransactionalResource之后，ApplicationServer要做如下两件事情。

a) ApplicationServer从TransactionalResource中取得XAResource交给TransactionManager。TransactionManager开始通过获得的这个XAResource与RM进行交互。实际上，现在TransactionManager只是调用XAResource的start(xid)方法通知RM开始记录。

b) ApplicationServer然后再把与XAResource属于同一个TransactionalResource的Connection传给客户端应用程序使用，然后客户端应用程序就可以使用ApplicationServer传给的Connection进行数据访问操作了。

(4) 客户端应用程序数据访问操作完成，关闭之前ApplicationServer传给的Connection，ApplicationServer在感知到Connection被关闭之后，会通知TransactionManager，TransactionManager则调用与这个Connection属于同一个TransactionalResource的XAResource的end(xid)方法结束事务记录。如果在当前分布式事务期间还有使用其他RM进行的数据操作，ApplicationServer以几乎同样的方式从RM的适配器那里获取TransactionalResource类似的对象，然后协调TransactionManager重复余下的工作。

(5) 当客户端通过UserTransaction或者TransactionManager的相应方法要求结束事务的时候，ApplicationServer就会通知TransactionManager使用两阶段提交协议提交当前事务：

a) TransactionManager调用XAResource的prepare(xid)方法通知各个RM准备提交事务；

b) 如果各个XAResource回答全部OK，TransactionManager调用XAResource的commit(xid)方法通知各个RM最终提交事务。

可见，各个RM确实参与到了JTA TransactionManager所管理的分布式事务中，只不过，参与的过程由ApplicationServer对客户端应用程序屏蔽了。之所以要求客户端应用程序通过应用服务器的JNDI获取DataSource等资源，是因为只有使用ApplicationServer公开的与XAResource绑定到同一TransactionalResource的Connection，才可以保证客户端应用程序所做的所有数据访问操作能够加入ApplicationServer所协调的分布式事务中（如图21-6所示）。

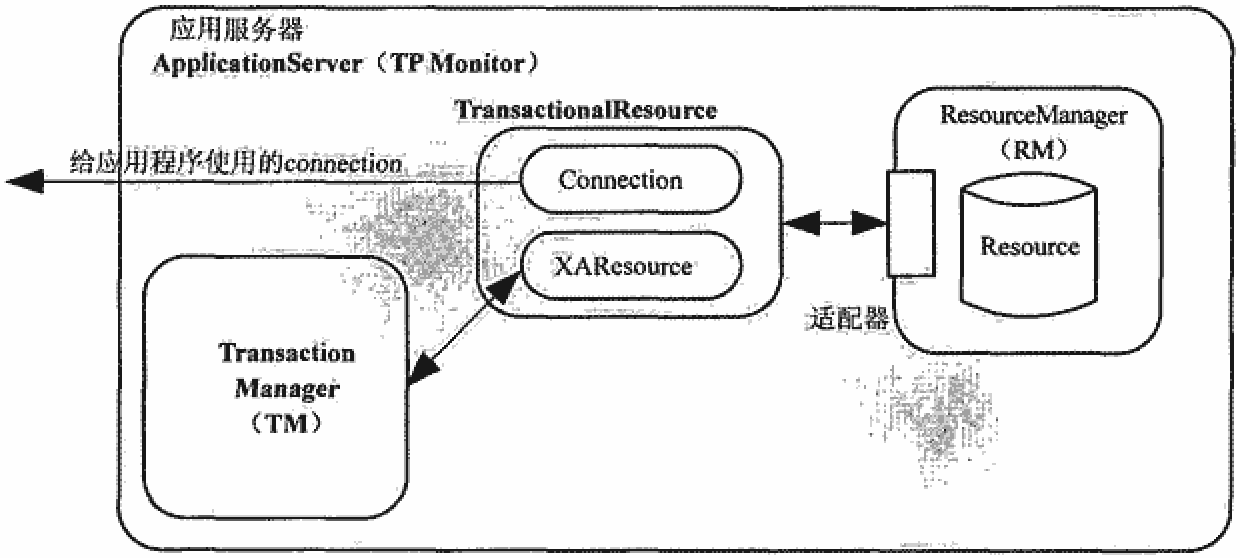

图21-6　ApplicationServer公开的Connection

ApplicationServer为客户端应用程序公开与当前分布式事务相关的Connection的方式，就是实现一个DataSource，然后把该DataSource绑定到JNDI。这样，客户端应用程序就可以通过从JNDI取得的DataSource中获取与事务相关的Connection了。如果使用本地定义的DataSource，因为它与当前分布式事务不发生任何关系，所以，也就根本不可能参与到分布式事务中去。

不过，如果我们使用的JTA实现不是相应的ApplicationServer提供的，比如，可以独立使用的Atomikos或者JOTM等JTA实现，要求我们从应用服务器的JNDI服务取得相应的DataSource这一前提也是不成立的。这时，我们直接使用各个JTA产品提供的DataSource封装类进行数据访问即可，与ApplicationServer屏蔽掉RM与TransactionManager之间的关系一样，这些产品也有与ApplicationServer完成同样工作的角色，为我们关联具体的RM与当前产品的TransactionManager。代码清单21-8中是在Spring中使用Atomikos的典型配置方式。

代码清单21-8　使用Atomikos进行分布式事务管理的IoC容器中的配置示例

```
<bean id="datasource1" ➥
class="com.atomikos.jdbc.SimpleDataSourceBean" init-method="init" destroy- ➥
method="close">
    <property name="uniqueResourceName" value="XADBMS_ONE"/>
    <property name="xaDataSourceClassName" value="COM.FirstSQL.Dbcp.DbcpXADataSource"/>
    <property name="xaDataSourceProperties" value="user=username;portNumber=8000"/
    <property name="exclusiveConnectionMode" value="true"/>
</bean>

<bean id="datasource2" ➥
class="com.atomikos.jdbc.SimpleDataSourceBean" init-method="init" destroy- ➥
```

```
method="close">
    <property name="uniqueResourceName" value="XADBMS_TWO"/>
    <property name="xaDataSourceClassName" value="COM.FirstSQL.Dbcp.DbcpXADataSource"/>
    <property name="xaDataSourceProperties" value="user=username;portNumber=8000"/
    <property name="exclusiveConnectionMode" value="true"/>
</bean>

<bean id="atomikosTransactionManager" ➥
class="com.atomikos.icatch.jta.UserTransactionManager" init-method="init" ➥
destroy-method="close">
    <property name="forceShutdown" value="true"/>
</bean>

<bean id="atomikosUserTransaction" class="com.atomikos.icatch.jta.UserTransactionImp">
    <property name="transactionTimeout" value="200"/>
</bean>

<bean id="springTransactionManager" class="org.springframework.transaction. ➥
jta.JtaTransactionManager">
    <property name="transactionManager" ref="atomikosTransactionManager"/>
    <property name="userTransaction" ref="atomikosUserTransaction"/>
</bean>

<bean id="dao" class= "...">
    <property name="dataSource" ref="datasource"/>
</bean>
```

我们需要告知Spring的JtaTransactionManager使用Atomikos的usertransaction和transactionManager实现。而至于Atomikos的userTransaction和transactionManager到底如何与RM（数据库、消息队列以及其他）进行交互，那就是Atomikos的事情了。

> 注意 不要被Atomikos的SimpleDataSourceBean的名字给迷惑了。另外，只要应用程序使用的数据库提供了有XA支持的数据库驱动，我们就可以通过SimpleDataSourceBean来配置需要的支持XA的DataSource，但各个参数的设置需要参考相应驱动程序的文档，比如xaDataSourceClassName以及xaDataSourceProperties属性。
> 多个Atomikos的SimpleDataSourceBean存在的情况下，它们对应的uniqueResourceName必须是不同的！

最后的内容是专门为Spring的JtaTransactionManager的，Spring 2.5版本发布后，在XSD的配置中，可使用tx命名空间下专门为JtaTransactionManager提供的配置元素来简化其配置，如下所示：

```
<tx:jta-transaction-manager/>
```

当然，如果需要，也可以通过内嵌的<property>指定使用自定义的usertransaction或者tran-sactionmanager。

21.4 小结

作为Spring事务抽象框架的扩展篇，本章主要挖掘了事务抽象框架中有关ThreadLocal的实践、策略模式的实践，以及分布式事务管理的简单分析这三点内容。我希望以这些内容为砖，能够引出更多的玉。

第六部分

Spring的Web MVC框架

在学习本部分内容之前，我希望你已经对Servlet以及JSP有了基本的了解。如果情况并非如此，那不妨先放下这部分内容，回头参阅一些Servlet以及JSP的书籍或相关资料。在你有了这部分知识储备之后，再继续本章的旅程。

本部分内容

- 第22章　迈向Spring MVC的旅程
- 第23章　Spring MVC初体验
- 第24章　近距离接触Spring MVC主要角色
- 第25章　认识更多Spring MVC家族成员
- 第26章　Spring MVC中基于注解的Controller
- 第27章　Spring MVC之扩展篇

第22章 迈向Spring MVC的旅程

本章内容

- Servlet独行天下的时代
- 繁盛一时的JSP时代
- Servlet与JSP的联盟
- 数英雄人物，还看今朝

"子曰：温故而知新"，如果简单回顾一下整个Java平台上的Web开发历程，将极大有助于我们理解当前各个Web开发框架存在的背景和先进性，最主要的是有助于我们平滑过渡到Spring MVC的世界中去。所以，不妨让我们从最初的Servlet独行天下的时代说起。

22.1 Servlet 独行天下的时代

话说Servlet是当年Java平台上第一个用于Web开发的技术。相对于CGI（Common Gateway Interface）时代来说，Servlet的提出是一个很大的进步，它运行于Web容器（Web Container）之内，提供了Session和对象生命周期管理等功能。最主要的，Servlet依然是Java类，从中直接访问Java平台的各种服务，并使用相应的API支持是很自然的事情，比如调用业务对象，使用JDBC进行数据访问等。

Servlet本身并非万能的，它的强项在于无缝地衔接业务对象与Web层对象之间的调用，以及二进制显示内容的输出等。但在当时开发人员只有Servlet这一种技术可以选择的时候，或许是不得不，也或许是盲从，又或许根本就是图一时的省事儿，单一的Servlet被赋予了过多的使命，从而导致了开发过程中出现的一系列的弊病。最常见的，就是称为"神奇Servlet（Magic Servlet）"的普遍存在。在"神奇Servlet"中，开发人员会将各种逻辑混杂于一处，包括流程控制逻辑、视图显示逻辑、业务逻辑、数据访问逻辑等，这就造成了后期系统的难以维护等一系列问题。

我们不妨看一段模拟当年的Web应用中的Servlet实现代码，见代码清单22-1。

代码清单22-1　模拟神奇"Servlet"的代码示例

```
public class MockMagicServlet extends javax.servlet.http.HttpServlet implements ➥
javax.servlet.Servlet {
    private static final long serialVersionUID = 3122666952706765103L;

    protected void doGet(HttpServletRequest request, HttpServletResponse response) ➥
    throws ServletException, IOException {
        doPost(request, response);
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse response) ➥
```

```
    throws ServletException, IOException {
        String parameter1 = request.getParameter("paramName1");
        String parameter2 = request.getParameter("paramName2");
        // ...

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<HTML>");
        out.println("<HEAD><TITLE>Page Title<TITLE><HEAD>");
        out.println("<BODY>");
        out.println("<table width=\"200\" border=\"1\">");
        out.println("<tr><td>Title1</td><td>Title2</td><td>Title3</td></tr>");

        String SQL = "select * from SomeTable where Column1=? and Column2=?";
        Connection con = getConnection();
        try {
            PreparedStatement ps = con.prepareStatement(SQL);
            ps.setString(1, parameter1);
            ps.setInt(2, Integer.parseInt(parameter2));
            // 其他可能的参数设置……
            ResultSet rs = ps.executeQuery();
            while(rs.next())
            {
                out.println("<tr>");
                out.println("<td>"+rs.getString(1)+"</td>");
                out.println("<td>"+rs.getString(2)+"</td>");
                // ...
                out.println("</tr>");
            }
            rs.close();
            ps.close();
        } catch (SQLException e) {
            e.printStackTrace(); // 不要这样做
        }
        closeConnection(con);
        out.println("</table>");
        out.println("<BODY>");
        out.println("<HTML>");
        ...
        out.close();
    }
    ...
}
```

代码的可读性极差，这一点先放一边不说，单就“数据访问逻辑和业务处理逻辑与对应的视图渲染逻辑相互混杂”这一点来说，就已经让今天的我们觉得该Servlet的实现是如此的不堪入目了。没有将相应的关注点进行明确的分离，直接导致相应的逻辑无法重用，进而造成后期系统难以维护。那么，我们有没有办法来重构这段代码，以使得它结构良好，易于维护呢？

实际上，我们只要将相应的逻辑以独立的形式剥离出来，避免这些逻辑之间的混杂，就应该能够得到一个结构清晰的应用。对于使用JDBC原始API进行数据访问的代码逻辑来说，有了之前Spring数据访问相关内容作为基础，对其进行重构应该是一件易事。假设你已经能够将这部分逻辑剥离到相应的数据访问对象，并提供了相应的MockServletService封装一系列的数据访问逻辑、事务管理逻辑等，那么，重构后的MockMagicServlet的代码看起来如代码清单22-2所示。

代码清单22-2　重构后的MockMagicServlet代码片段摘录示例

```
...
protected void doPost(HttpServletRequest request, HttpServletResponse response) ➥
throws ServletException, IOException {
    String parameter1 = request.getParameter("paramName1");
    String parameter2 = request.getParameter("paramName2");
    // ...
    List<InfoBean> infoList = MockServletService.query(parameter1,parameter2);

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<HTML>");
    out.println("<HEAD><TITLE>Page Title<TITLE><HEAD>");
    out.println("<BODY>");
    out.println("<table width=\"200\" border=\"1\">");
    out.println("<tr><td>Title1</td><td>Title2</td><td>Title3</td></tr>");

    for(InfoBean infoBean : infoList)
    {
        out.println("<tr>");
        out.println("<td>"+infoBean.getColumn1()+"</td>");
        out.println("<td>"+infoBean.getColumn2()+"</td>");
        // ...
        out.println("</tr>");
    }
    out.println("</table>");
    out.println("<BODY>");
    out.println("<HTML>");
    ...
    out.close();
}
...
```

哇噢，清爽多了，不是吗？不过，我们依然没有摆脱那些烦人的out.println，而且，对于还处于懵懂状态的Servlet开发时代来说，这些out.println可不只是烦人而已。

- "神奇Servlet"的存在，并不意味着一个Web应用程序中只存在一个Servlet实现。实际上，Servlet时代之初，我们更多是使用"一个Servlet对应处理一个Web请求"的方式。对于简单的Web应用来说，这种用于生成视图的out.println语句，分散的程度看起来还不算夸张。可是，随着应用规模的扩展，试想一下，开发和维护这些out.println的工作量将是何等的恐怖？
- 我们的代码示例只是给出了一个简单的视图渲染逻辑。可是，随着页面逻辑的膨胀，要维护这么一堆几乎无法"纵观全局"的out.println，我们这些开发人员该怎么办呢？暂且不谈开发人员大都不熟悉(X)HTML等页面标记语言，就算熟知，要在浩瀚的out.println中寻找要更改的位置，并且保证更改过程中不会造成之前的显示逻辑的破坏，又是谈何容易啊！即使是使用Jakarta ECS（Element Construction Set）这样的类库，我想也不会减去多少维护这些out.println的痛苦。
- 因为视图逻辑是以Java代码的形式写死到Servlet中的，所以如果视图逻辑需要变动的话，我们就得更改Servlet的代码并重新编译。开发人员或许会说视图逻辑不归我们管，那是美工和前台开发人员的工作，可是，写死到Java代码中的视图逻辑，你又能逃脱了干系不成？

令人遗憾的是，仅靠Servlet一人之力，我们无法解决视图逻辑与Servlet紧密耦合的问题。好消息是，这个时候有了JSP前来救驾。

> **提示**　请稍微关注以上的Servlet代码。当然，这样说，并不是因为它臻于完美，而是因为不管这之后的Web应用的开发如何演化，都将以该Servlet为基础进行，不信的话，继续往下看。

22.2 繁盛一时的 JSP 时代

为了能够将Servlet中的视图渲染逻辑以独立的单元抽取出来，我们通常使用“模板化”（templating）的方法。JSP的提出，成为Java平台上开发Web应用程序事实上的模板化视图标准。

有了JSP的加入，我们就可以将原先不得不在Servlet中通过`out.println`语句输出的视图渲染逻辑，抽取到.jsp后缀名的模板文件中。这样，我们就有了如代码清单22-3所示的视图模板定义。

代码清单22-3　抽取到JSP的视图逻辑模板定义代码示例

```
<%@ page contentType="text/html" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head>
<meta content="text/html; charset=utf-8" http-equiv="Content-Type" />
<title>Page Title</title>
</head>
<body>
    <table border="1">
        <tr><td>Title1</td><td>Title2</td><td>Title3</td></tr>
        <c:forEach items="${infoList}" var="infoBean">
            <tr>
                <td>${infoBean.column1}</td>
                <td>${infoBean.column2}</td>
                <td>${infoBean.column3}</td>
            </tr>
        </c:forEach>
    </table>
</body>
</html>
```

现在，由JSP专职负责视图的渲染工作，我们的`MockMagicServlet`得以进一步的解脱。现在，`MockMagicServlet`的主要实现逻辑看起来如代码清单22-4所示。

代码清单22-4　进一步重构后的`MockMogicServlet`实现代码片段摘录示例

```
...
protected void doPost(HttpServletRequest request, HttpServletResponse response) ➥
throws ServletException, IOException {
    String parameter1 = request.getParameter("paramName1");
    String parameter2 = request.getParameter("paramName2");
    // ...
    List<InfoBean> infoList = MockServletService.query(parameter1,parameter2);
    request.setAttribute("infoList", infoList);
    forward(request,response,"view.jsp");
}
protected void forward(HttpServletRequest request, HttpServletResponse response,String ➥
viewPath) throws ServletException, IOException {
    RequestDispatcher.requestDispatcher = ➥
    getServletContext().getRequestDispatcher(viewPath);
    requestDispatcher.forward(request, response);
}
...
```

啊哈，看起来我们离成功仅是一步之遥啦！但是，不好意思，在此之前，我们还需要经历一段曲折的日子。

实际上，如果当初我们的开发人员或者是“技术布道者”能够严格地界定JSP的基本使命的话，我们早就迈入了Web MVC的世界了。不过，如果我们更愿意“怨天尤人”，也可以把责任推卸给JSP

本身。我们都知道，JSP与其他模板技术有一个主要的区别，那就是，它最终是编译为Servlet来运行的，这一层关系使得JSP拥有比其他通用模板技术更大的能量。

- 我们可以直接在JSP中编写Java代码，通过Scriptlet，只要愿意，任何的应用程序逻辑几乎都能够融入JSP广阔的“胸襟”。我还清晰地记得当初自己写下的第一个Web应用程序是什么样子，整个应用几乎都是JSP文件组成，几乎什么逻辑全部通过Scriptlet编写到JSP文件中，甚至于数据库连接代码。我想，这只是一个缩影，实际上，那几年无论是技术社区还是技术书籍，倡导的也都是类似理念。比如介绍JSP的书籍，即使不应该放到JSP内的逻辑，也全都写入JSP的实例中，使得你不上当都难。可是现在回头看看，如果说我们之前创造了“神奇Servlet”，那么，现在，我们创造的则是神奇JSP了。相应的弊病，就不多说了。
- 使用Servlet处理Web请求，我们需要在web.xml文件中，注册相应的请求URL与具体的处理Servlet之间的一个映射关系。之前说过，最初阶段，我们是一个Web请求对应一个Servlet进行处理的，所以web.xml中就过多地充斥着与代码清单22-5中类似的映射配置信息。

代码清单22-5　web.xml中Servlet的映射关系配置示例

```
<!-- Servlet定义-->
<servlet>
    <servlet-name>MockMagicServlet</servlet-name>
    <servlet-class>package.name.MockMagicServlet</servlet-class>
</servlet>
<servlet>
    <servlet-name>OtherServlet</servlet-name>
    <servlet-class>package.name.OtherServlet</servlet-class>
</servlet>
...

<!-- 映射关系定义 -->
<servlet-mapping>
    <servlet-name>MockMagicServlet</servlet-name>
    <url-pattern>/requestPath1</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>OtherServlet</servlet-name>
    <url-pattern>/requestPath2</url-pattern>
</servlet-mapping>
...
```

使用JSP的话，则可以省却这些烦琐，直接通过链接就可以，无需任何配置。这也助长了“超频”使用JSP的“歪风”，直接使用JSP替代Servlet处理Web请求。对于简单的应用，或许几个页面的流程关系还理得清楚，一旦应用规模上来了，分散于各个JSP文件中的流程控制信息，无异于一张杂乱无章的网，令人理不清，道不明，更不用说易于管理和维护了。而且，将原本由Servlet处理的Web流程逻辑纳入JSP的职权，我们又进一步帮助了神奇JSP的尽快诞生。

本该行使简单的视图渲染功能的JSP，现在已经完全替代了Servlet而一统天下了，也就是说，我们现在不需要MockMagicServlet了，取而代之的，就是我们原先设想用于剥离试图渲染逻辑的JSP。而现在，它自身已经不只是一个视图模板了（见代码清单22-6）。

代码清单22-6　包揽全部职能的JSP代码示例

```
<%@page import="java.sql.DriverManager"%>
<%@page import="java.sql.Connection"%>
<%@page import="java.sql.PreparedStatement"%>
```

```
<%@page import="java.sql.ResultSet"%>
<%@page import="java.sql.SQLException"%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<table>
<tr><td>Title1</td><td>Title2</td><td>Title3</td></tr>
<%
String parameter1 = request.getParameter("paramName1");
String parameter2 = request.getParameter("paramName2");

String SQL = "select * from SomeTable where Column1=? and Column2=?";
Class.forName("your.driver.class.name");
Connection con = DriverManager.getConnection("serverAddress");
try {
    PreparedStatement ps = con.prepareStatement(SQL);
    ps.setString(1, parameter1);
    ps.setInt(2, Integer.parseInt(parameter2));
    // 其他可能的参数设置……
    ResultSet rs = ps.executeQuery();
    while(rs.next())
    {
%>
<tr>
    <td><%=rs.getString(1)%></td>
    <td><%=rs.getString(2)%></td>
    <td><%=rs.getString(3)%></td>
</tr>
<%
    }
    rs.close();
    ps.close();
} catch (SQLException e) {
    e.printStackTrace(); // 不要这样做
}
con.close();
%>
</table></body>
</html>
```

可是，历史是相似的。当初为了解决“神奇Servlet”的问题，而被迫分离应用的各种关注点，我们是否也注意到了现在的JSP又以相似的步伐重蹈“神奇Servlet”的覆辙呢？ 答案是肯定的，不光我们注意到了，SUN公司也注意到了，所以，这也促使了JSP Model 1的诞生（见图22-1）。

在JSP独大的世界中引入JavaBean，通过JavaBean对相关业务逻辑的封装，我们完成了初步的关注点分离。不过，JSP Model 1的提出并没有进一步地限定JSP的基本职责，本该一心关注视图渲染逻辑的JSP，现在依然紧攥着本该是Servlet强项的Web流程管理功能不放。看来，革命尚未成功，我们依然需要努力啊！

注意 单独使用JSP阶段的Web开发，还存在许多的弊端，你可以从Rod Johnson的*Expert One-on-One J2EE Design and Development*一书中了解更多详情。

另外，JSP Model 1也有其先进性，我们可以用它快速的构建Web应用程序的原型，但是，切记，不要以这种架构用于实际的生产环境。

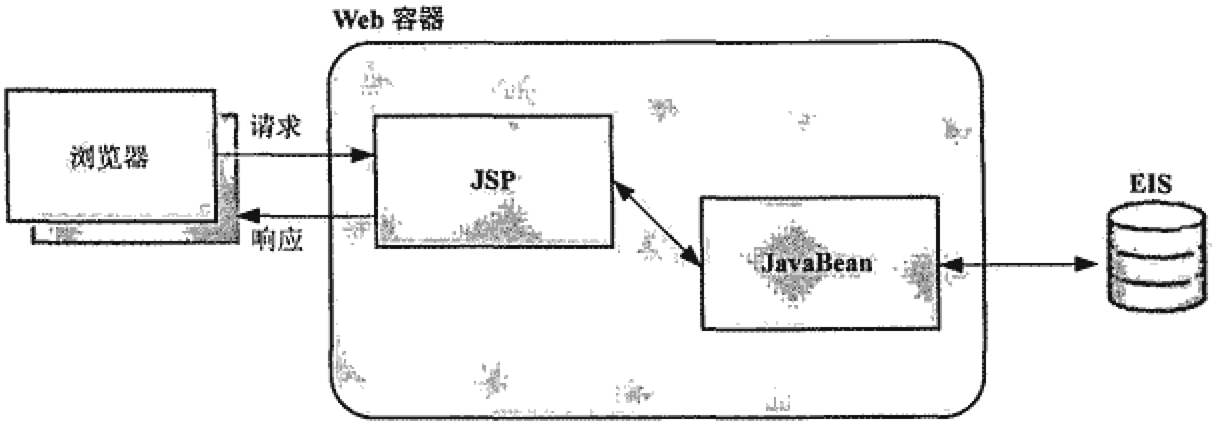

图22-1　JSP Model 1示意图

22.3 Servlet 与 JSP 的联盟

JSP的不良诱惑使得我们走上了歧路，本已经近在咫尺的良好架构，在经历了一段尘封的岁月之后，又重现光芒。让我们回到JSP时代的开始，在那里，我们让JSP做为视图模板而存在。不管它有多大的能耐，我们只让它负责视图的渲染工作，这样，对于JSP来说，只需要页面开发人员或者说表现层（presentation layer）开发人员来负责和管理即可。而已经剥离了视图渲染逻辑Servlet，现在只负责请求处理流程的控制以及与业务层的交互，这部分工作则由Java开发人员来负责。至此，我们不仅将各个关注点清晰地分离出来，而且也分离了Java开发人员与前台开发人员之间的职责。而后者对于一个复杂并且需要多人协作的团队来说，是至关重要的。

通过结合Servlet和JSP，并且引入封装业务层逻辑的JavaBean，我们得到了JSP Model 1的升级版架构，即JSP Model 2，见图22-2。

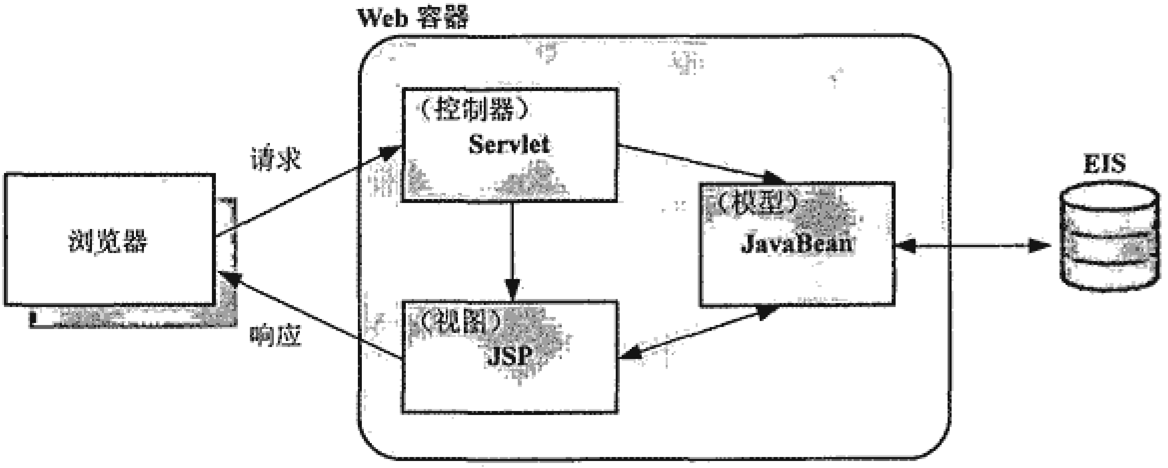

图22-2　JSP Model 2示意图

在Model 2中，与我们重构过程结果所展示的一样，由Servlet来管理处理流程，由JSP来负责视图的渲染，并由JavaBean封装业务逻辑并负责与数据层进行交互。JSP Model 2的提出可以说是Java平台Web应用开发过程中一个里程碑，它促进了Web开发领域至今一直沿用的MVC设计模式的广泛应用。

虽然JSP Model 2已经具备了使用MVC模式实现的Web应用架构的雏形，但并非严格意义上的MVC。为了搞清楚其间的差别，我们先来简单回顾一下MVC模式以及模式中涉及的几个组件，见图22-3。

MVC（Model-View-Controller，模型-视图-控制器）在当今Java界尤其是Web开发领域已经是耳熟能详的一个名词了。如图22-3所示，最初意义上的MVC模式中，各个组件的作用如下所述。

- 控制器负责接收视图发送的请求并进行处理，它会根据请求条件通知模型进行应用程序状态的更新，之后选择合适的视图显示给用户。
- 模型通常封装了应用的逻辑以及数据状态。当控制器通知模型进行状态更新的时候，模型封装的相应逻辑将被调用。执行完成后，模型通常会通过事件机制通知视图状态更新完毕，从而视图可以显示最新的数据状态。
- 视图是面向用户的接口。当用户通过视图发起某种请求的时候，视图将这些请求转发给控制器进行处理。处理流程流经控制器和模型之后，最终视图将接收到模型的状态更新通知，然后视图将结合模型数据，更新自身的显示。

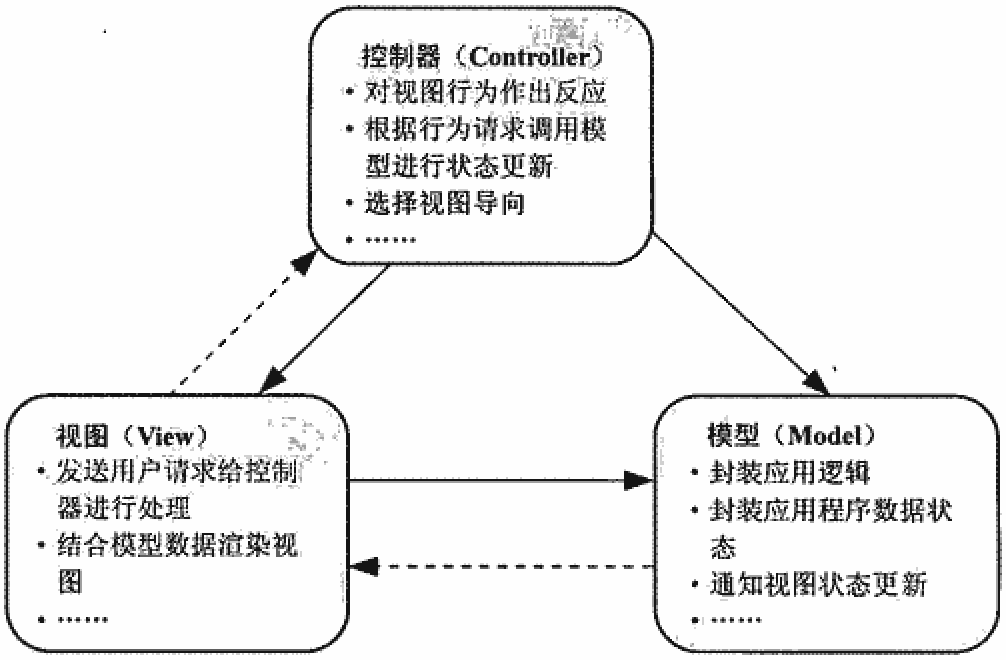

图22-3 MVC模式示意图

可见，最初意义上的MVC模式，在视图与模型间的数据同步工作是采用从模型**Push**到视图的形式完成的。而对于Web应用来说，局限于所用的协议以及使用场景，无法实现从模型**Push**数据到视图这样的功能。所以，我们只能对MVC中的组件的最初作用定义做适当的调整，由控制器与模型进行交互，在原来通知模型更新应用程序状态的基础上，还要获取模型更新的结果数据，然后将更新的模型数据一并转发给视图。也就是说，我们现在改由控制器从模型中**Pull**数据给视图，这种意义上的MVC称为Web MVC，也就是现在大多说Web开发框架所使用的架构模式。

实际上，JSP Model 2已经十分接近Web MVC架构了，但是，在真正步入Web MVC应用框架时代之前，我们还是来看一下JSP Model 2在具体的应用过程中存在哪些问题，我们又是如何解决这些问题，并进而促成Web MVC应用程序框架的广泛应用的吧！从JSP Model 2架构的示意图（图22-3）上，我们可以看到，Servlet是作为控制器的角色存在的，但是，该架构示意图并没有进一步规定，具体应用中到底是只需要一个控制器，还是使用多个控制器，这就造成如下两种情况。

- **Web应用程序中使用多个Servlet作为控制器。**这实际上也是从最初Servlet步入Java平台Web开发领域后使用最多的模式，即一个Servlet对应一个Web请求的处理。以这种方式进行的开发实践表明，我们需要为每个请求处理流程都定义一个Servlet，并借助Web容器的URL映射匹配能

力来解决Web请求到具体的处理Servlet的映射。自然，我们就需要在web.xml配置文件中，为每个Servlet都提供定义并添加URL映射。随着应用规模的增加，web.xml的体积将愈加庞大。最主要的，系统中的所有Web请求的处理流程将各自分散管理，没有一种集中管理的方式。这将不利于整个系统的开发和维护工作。所以，随着开发理念的更新， 这种方式逐步淡化出了我们的视野，并更多侧重于下面这种单一Servlet作为整个Web应用程序控制器的实践方式。

- **Web应用程序中使用单一Servlet作为集中控制器**。现在，所有的Web处理请求全部经由Web应用程序中定义的这个单一的Servlet控制器来进行。相对于原先的情况，请求的处理现在有了一个集中管理的位置，而且，也不用顾虑web.xml文件内容是否会因Web请求流程的增多而膨胀。不过，却遇到了新的问题，我们避免了web.xml文件的膨胀，却将这种膨胀变相地带到了Servlet控制器类中，如下所述。
  - 因为现在所有的Web请求都映射到了集中的Servlet控制器来处理，所以，控制器类需要自己来根据Web请求的URL信息进行分析，以判断处理流程的流向。显然，无法再借助Web容器的URL映射匹配能力来完成这个工作了。早期来说，这些逻辑都是硬编码到Servlet控制器中的。这些逻辑往往不能重用，而且最主要的，一旦写死，要调整URL映射的处理，就得修改Servlet控制器的代码并重新编译，灵活性和可扩展性根本无从谈起。
  - 分析完URL映射关系之后，Servlet控制器就可以根据结果来选择执行哪些处理流程，硬编码的问题再次出现了。那个时候的控制器Servlet类，大都是将处理流程和处理逻辑硬编码到自身，无论是流程分支的调整，还是具体每个分支的处理逻辑的调整，都不可避免地要对Servlet控制器的实现代码进行一番“或大或小”的手术。当然，下一个应用开始之后，这些处理流程的转发逻辑以及其他通用逻辑，是无法复用到下一个应用程序中的。

不过，情况并没有我们所看到的那么坏，引入合适的设计模式，可以避免早期单一Servlet控制器所面临的问题。所以，这种方式依然是我们比较倾向使用的控制器实践方式。

如我们所看到的，制约JSP Model 2发展的，就是将流程控制等通用相关逻辑进行硬编码的实践方式，这直接导致了JSP Model 2架构的不可重用性。每次启动新的Web应用程序的开发工作，通常又得从头编写Servlet控制器的URL分析，以及流程控制等Web层通用逻辑。这自然就促使我们寻找途径，去除架构中控制逻辑的硬编码，并尽可能地复用Web应用程序开发过程中的一些通用逻辑。而在JSP Model 2架构基础上发展起来的各种Web MVC应用框架，恰好顺应了历史的需求。现在，我们步入了各种Web应用程序框架盛行的时代。

22.4 数英雄人物，还看今朝

Web框架存在的意义在于，它们为Web应用程序的开发提供了一套可复用的基础设施，这样开发人员只需要关注特定于每个应用的逻辑开发工作，而不需要每次都重复那些可以统一处理的通用逻辑。当前的Web开发框架有如下两种类型。

- **请求驱动的Web框架**（request-driven framework）。又称为request/response框架（request/response framework）。顾名思义，这种框架是基于Servlet的请求/响应（request/response）处理模型构建的。这种类型的开发框架大都以Web MVC模式为指导，在JSP Model 2架构基础上“进化”而来。比如，几乎是整个Java平台Web开发框架事实标准的Struts框架，优雅轻便的WebWork等，以及我们稍后即将为你介绍的Spring MVC框架，都属于这种请求驱动的Web开发框架。
- **事件驱动Web开发框架**（event-driven web framework）。或许它的另一个名字基于组件的Web

开发框架（component-based framework）更好理解一些。这种框架采用与Swing等GUI开发框架类似的思想，将视图组件化，由视图中的相应组件触发事件，进而驱动整个处理流程。最初的Tapestry（http://tapestry.apache.org/）框架以及现在的JSF（Java Server Faces）框架都属于这一类。

Spring MVC属于请求驱动的Web框架，所以，我们要对这种类型的框架给予更多的笔墨。至于事件驱动的Web开发框架，如果你感兴趣，不妨自己做进一步深入的了解。

对于请求驱动的Web开发框架来说，它们大多是在JSP Model 2的基础上发展而来的。我们一定有一个问题，那就是，这些Web开发框架是如何解决JSP Model 2在实践中所存在的那些问题的？如前所述，在JSP Model 2中，我们更加倾向于使用单一Servlet作控制器的实践方式。实际上，现在的请求驱动的Web框架也大都如此，但是为了避免之前提到的一些问题，这些框架通常会结合Front Controller以及Page Controller模式[①]，对单一Servlet控制器做进一步的改进，对原先过于耦合的各种控制器逻辑进行逐步的分离。具体来说，就是由原来的单一Servlet作为整个应用程序的Front Controller。该Servlet接收到具体的Web处理请求之后，会参照预先可配置的映射信息，将待处理的Web处理请求转发给次一级的控制器（sub-controller）来处理，整个情形看起来如图22-4所示。

22

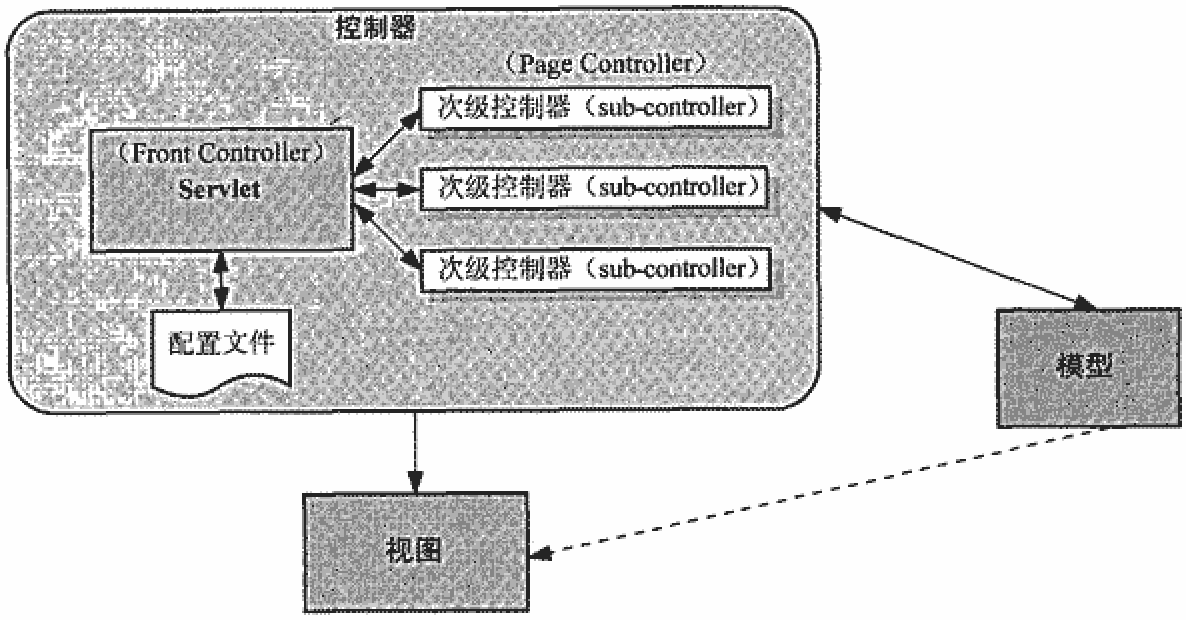

图22-4　引入Front Controller模式后的控制器实现结构

在控制器Servlet接收到Web处理请求之后，它会对Web请求的URL信息进行分析，然后根据分析结果，并参照外部可配置的流程规则，将当前Web请求转发给次一级的控制器类进行处理。现在，作为Front Controller的Servlet和次级控制器类共同组成了整个应用程序的控制器。原先单一的控制器Servlet通过将流程控制信息外部化，并分离具体的Web请求处理逻辑给次级控制器类进行处理的方式，瘦身为灵活而可复用的担当Front Controller的Servlet。有了这样的关注点分离之后，我们就极大地提高了整个Web应用中控制器逻辑的可复用性。

如果我们之前接触过Struts框架，对照一下以上的结构，将有助于更深入地理解该框架中各个组件所扮演的角色：

- ActionServlet是整个框架的Front Controller，负责分发具体的Web请求；

① 参照Martin Fowler的*Pattern of Enterprise Application Architecture*一书中有关这两种模式的介绍。

- Action是次级控制器，ActionServlet分发的具体Web请求将由被选中的Action来处理；
- 为了避免控制流程的硬编码，ActionServlet将从struts-config.xml中读取请求的URL与具体的Action之间的映射关系，进而正确地转发Web请求给相应的Action进行处理。

当然，各个Web框架中实现相应角色的具体实现类或许不同，但所要达到的目的是一样的。而且，有些框架还在以上实现结构的基础上，进一步细化了框架中可能复用的逻辑。在介绍完Spring MVC之后，你将发现一个设计更加完善的Web开发框架。

实际上，Spring MVC可以说是集之前各个Web框架的优点于一身，并且还有进一步的发展。虽然最初考虑到Struts的市场地位，我并不看好Spring MVC，但是，在深入了解它之后，我改变了之初的想法。相信当你更多地了解Spring MVC之后，也会深深地迷恋于此。话不多说，还是赶快开始我们的Spring MVC之旅吧！

22.5 小结

本章我们主要回顾了Java平台上以往的Web开发历程。我们从Servlet出现之初独当一面的时候开始，进而经历了JSP繁盛一时的时代，然后就是Servlet和JSP结盟，最终演化到现在Web开发框架盛行的年代。

有这些背景作为基础，接下来，我们将正式进入Spring MVC的领地，看一下Spring MVC框架到底有何独到之处。

第23章 Spring MVC初体验

本章内容

- 鸟瞰Spring MVC
- 实践出真知

我想每个人在接触一个新的Web框架时，都会问一个问题，这个框架的优势在哪里？或者说，它比其他框架好在哪里？在现在这个各种Web开发框架满天飞的年代，开发人员对于框架的选择更是尤其谨慎。我不想标榜Spring MVC比其他的Web框架有多好多好。如果你之前使用Struts开发，并且有足够的实践经验和过硬的技能，并且Struts也能够满足你的需要，那么，这种情况下换另一种新的开发框架或许不是一个好的主意。对于WebWork等框架也是同样的道理。但如果你是第一次选择Web开发框架，那么，考虑到以下几点，Spring MVC还是可以进入重点考虑之列的。

- 在Web层，或者确切点儿说，在框架的控制器的实现方面，Spring MVC对请求处理期间涉及的各种关注点进行了合理而完全的分离，并且明确设置了相应的角色用于建模并处理整个生命周期中的各个关注点，这包括：
  - 设置HandlerMapping用于处理Web请求与具体请求处理控制器之间的映射匹配；
  - 设置LocaleResolver用于国际化处理；
  - 设置ViewResolver用于灵活的视图选择等。

  而这些角色所处理的关注点，在此前的大部分Web开发框架中都没有明确划分，或者只是部分分离出来，而现在它们都是可扩展，可替换的。
- 从表现（presentation）层面来看，Spring运用逻辑命名视图（logical named view）策略，通过引入ViewResolver和View，清晰地分离了视图类型的选择和渲染（Render）与具体控制器之间的耦合，使得各种视图技术可以很容易地集成到Spring MVC框架中。不管是使用JSP/JSTL作为视图技术，还是使用Velocity/FreeMarker，甚至PDF/Excel等二进制格式的视图形式，要启用它们，现在只是简单的配置工作而已。
- Spring MVC独有的一个特色，得益于自己“出身名门”，它可以很容易并且很自然地得到自家“兄弟”的支持，从IoC容器提供的依赖注入支持，到Spring AOP的支持，以及数据访问层、事务管理层的支持等。Spring提供的良好的中间层支持，是之前许多Web开发框架都缺少的东西，而这个是Spring MVC与生俱来的，所以，即使在使用其他的Web开发框架，我们也应该寻求集成Spring框架在应用中间层的支持。

当然，我们可以从Spring参考文档中发现Spring MVC更多的特点。如果这些能够打动你，那么，余下的大餐就是为你准备的！

23.1 鸟瞰 Spring MVC

Spring MVC框架的处理控制器的实现策略，与其他的请求驱动的Web框架在总体思路上是相似的，就如我们之前所说的那样，通过引入Front Controller和Page Controller的概念来分离流程控制逻辑与具体的Web请求处理逻辑。org.springframework.web.servlet.DispatcherServlet就是Spring MVC框架中的Front Controller，它负责接收并处理所有的Web请求，只不过针对具体的处理逻辑，它会委派给它的下一级控制器去实现，即org.springframework.web.servlet.mvc.Controller，而org.springframework.web.servlet.mvc.Controller[①]则对应Page Controller的角色定义，具体关系如图23-1所示。

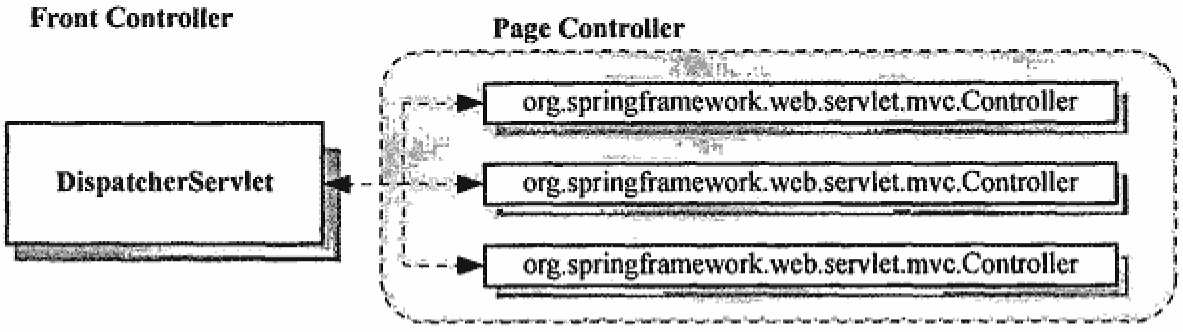

图23-1 Spring MVC中的Front Controller和Page Controller

当然，只有它们两个还远远称不上一个完整的Web开发框架，要使得整个的Web开发框架能够运作起来，我们还需要更多的角色相助。下面，让我带你走过DispatcherServlet这道长廊，并乘机一一介绍Spring MVC中的各位“掌权者”。

在我们开始DispatcherServlet的旅程之前，不妨先回顾一下控制器Servlet通常会做哪些工作。就以我们JSP Model 2架构中提到的MockMagicServlet为例（之前特意请你关注过它），它在处理Web请求的过程中所做的工作，我们可以简单归纳为如下三点。

- 获取请求信息，比如请求的路径、各种参数值等，如下所示：

```
String parameter1 = request.getParameter("paramName1");
String parameter2 = request.getParameter("paramName2");
// ...
```

- 根据请求信息，调用具体的服务对象处理具体的Web请求，如下所示：

```
List<InfoBean> infoList = MockServletService.query(parameter1,parameter2);
```

- 处理完成后，将要在视图中显示的模型数据通过request进行传递，最后通过Request-Dispatcher选择具体的jsp视图并显示，如下所示：

```
request.setAttribute("infoList", infoList);
forward(request,response,"view.jsp");
```

有了以上MockMagicServlet处理流程的剖析作为基础，现在我们再来看DispatcherServlet在整个Web请求处理过程中所做的事情，就会发现，DispatcherServlet在实现上并没有太多不同或者说改变，唯一的差异或者改变是，DispatcherServlet将各项工作细化并分离给了较为独立的

① 为了简化并且避免MVC模式中的C与Spring提供的org.springframework.web.servlet.mvc.Controller相互混淆，我们将会把org.springframework.web.servlet.mvc.Controller简称为Spring Controller。

角色来完成。

DispatcherServlet的处理流程[①]可以简单概括如下。

1. HandlerMapping先生（Web请求的处理协调人）

既然DispatcherServlet是整个框架的FrontController，当将它注册到web.xml时，就注定了它要服务于规定的一组Web请求的命运，而不是单独的一个Web请求。代码清单23-1是DispatcherServlet注册到web.xml配置文件的通常示例。

代码清单23-1 DispatcherServlet在web.xml中的配置代码示例

```
<servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>org.springframework.Web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>2</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>
```


不能够像“一个Web请求对应一个Servlet”那样获取Web容器对URL映射匹配的支持，DispatcherServlet现在只好自己来处理具体的Web请求和具体的处理类之间的映射关系匹配了。

对于Web请求与具体的处理类之间的映射匹配来说，具体的处理方式或者说策略可能多种多样，完全可能随着应用程序，甚至每个具体的Web请求的不同而发生变化，如下所述。

- 最常见的就是，那种“掐头去尾”的处理方式，将Web请求的URL路径去除前面的上下文路径（context path）和最后的扩展名，取最终剩下的路径信息，作为匹配的结果。比如，如下代码将最终以resource作为匹配结果。

  ```
  http://www.nosuchname.com/yourapp/resource.html
  ```

- 以Web请求的URL中存在的某个参数的值作为匹配的标准。比如，当发现请求的路径中存在controller这个参数的话，其值就会被作为匹配结果用来调用具体的处理类，对于如下所示的URL：

  ```
  http://www.nosuchname.com/yourapp/resource.html?controller=yourController
  ```

  匹配的结果就是yourController。

- 以cookie或者session中的某些信息作为匹配标准。比如，针对某个客户的Web请求，全部转发给一个处理类进行处理。

甚至于，结合Ruby On Rails的理念，我们在开发中规定一些惯例或者说约定，然后以这些惯例或者约定来解析Web请求的URL路径信息，以获取具体的处理类匹配。可见，要将映射匹配的逻辑写死到DispatcherServlet，是无法有效扩展的，而且匹配的方式也可能随着需求而变化。所以，Spring MVC为了能够灵活地处理映射的匹配，引入了org.springframework.Web.servlet.HandlerMapping来专门管理Web请求到具体的处理类之间的映射关系。在Web请求到达DispatcherServlet之后，Dispatcher将寻求具体的HandlerMapping实例，以获取对应当前Web请求的具体处理类，即org.springframework.web.servlet.Controller。

① 该处理流程的描述主要是为了便于读者理解，并不十分确切，DispatcherServlet实际的处理流程要关注更多的细节，但在此基础上，稍后我们将扩展它，以便读者可以循序渐进地全面掌握Spring MVC框架。

2. org.springframework.Web.servlet.Controller（Web请求的具体处理者）

org.springframework.Web.servlet.Controller是对应DispatcherServlet的次级控制器，它本身实现了对应某个具体Web请求的处理逻辑。在我们所使用的HandlerMapping查找到当前Web请求对应哪个org.springframework.web.servlet.Controller的具体实例之后，DispatcherServlet即可获得HandlerMapping所返回的结果，并调用org.springframework.web.servlet.Controller的处理方法来处理当前的Web请求。

org.springframework.Web.servlet.Controller的处理方法执行完毕之后，将返回一个org.springframework.Web.servlet.ModelAndView实例，ModelAndView包含了如下两部分信息。

- 视图的逻辑名称（或者具体的视图实例）。DispatcherServlet将根据该视图的逻辑名称，来决定为用户显示哪个视图。
- 模型数据。视图渲染过程中需要将这些模型数据并入视图的显示中。

有了ModelAndView所包含的视图与模型二者的信息之后，DispatcherServlet就可着手视图的渲染工作了。

3. ViewResolver和View（视图独立战争的领导者）

按照MockMagicServlet的处理流程，我们已经走到了最后一步，即选择并转到最终的JSP视图文件，即如下代码所展示的逻辑：

```
request.setAttribute("infoList", infoList);
forward(request,response,"view.jsp");
```

但是，对于一个Web框架来说，我们是不可以这么简单处理的。为什么呢？不要忘了，现在可用的视图技术可不只JSP一家，Velocity、Freemarker等通用的模板引擎，都可以帮助我们构建相应的视图，而它们是不依赖于request对象来传递模型数据的，甚至，我们也不只依赖JSP专用的RequestDispatcher来输出最终的视图。否则，我们也没有必要通过ModelAndView来返回视图以及模型数据了，直接在org.springframework.web.servlet.Controller内部完成视图的渲染工作就可以了。不是吗？鉴于此，Spring提出了一套基于ViewResolver和View接口的Web视图处理抽象层，以屏蔽Web框架在使用不同的Web视图技术时候的差异性。

那么，Spring MVC是如何以统一的方式，将相同的模型数据纳入不同的视图形式并显示的呢？实际上，撇开JSP使用的RequestDispatcher不谈，Servlet自身就提供了两种最基本的视图输出方式。不知道你还是否记得最初的out.println？基本来说，我们要向客户端输出的视图类型，可以分为文本和二进制两种方式，比如JSP/JSTL、Velocity/Freemarker等最终的输出结果都是以(X)HTML等标记文本形式表现的，而PDF/Excel之类则属于二进制内容行列。对于这两种形式的视图内容的输出，Servlet自身公开给我们的HttpServletResponse已经足够可以应付了。代码清单23-2展示了HttpServletResponse是如何帮助我们处理这两种形式的视图输出的。

代码清单23-2　通过HttpServletResponse输出视图数据代码示例

```
// 使用Servlet输出标记文本视图
String markupText = ...;
PrintWriter writer = response.getWriter();
writer.write(markupText);
writer.close();
...
// 使用Servlet输出二进制格式视图
byte[] binaryContext = ...;
ServletOutputStream out = response.getOutputStream();
```

```
out.write(binaryContext);
out.flush();
out.close();
```

在HttpServletResponse可以同时支持文本形式和二进制形式的视图输出的前提下，我们只要在最终将视图数据通过HttpServletResponse输出之前，借助于不同的视图技术API，并结合模型数据和相应的模板文件，就能生成最终的视图结果，如下伪代码所示：

```
获取模型（Model）数据；
获取视图模板文件（比如*.jsp、*.vm、*.fm、*.xls等）；
结合视图模板和模型数据，使用相应的视图技术API生成最终视图结果；
将视图结果通过HttpServletResponse输出到客户端；
完成！
```

这样，不管最终生成的视图内容如何，我们都可以以几乎相同的方式输出它们。但唯一的问题在于，我们不可能将每个视图的生成代码都纳入DispatcherServlet的职权范围，毕竟，每种视图技术的生成代码是不同的，而且所使用的视图技术也可能随着具体环境而变化。Spring MVC通过引入org.springframework.web.servlet.View接口定义，来统一地抽象视图的生成策略。之后，DispatcherServlet只需要根据Spring Controller处理完毕后通过ModelAndView返回的逻辑视图名称查找到具体的View实现，然后委派该具体的View实现类来根据模型数据，输出具体的视图内容即可。Spring的View抽象策略示意图如图23-2所示。

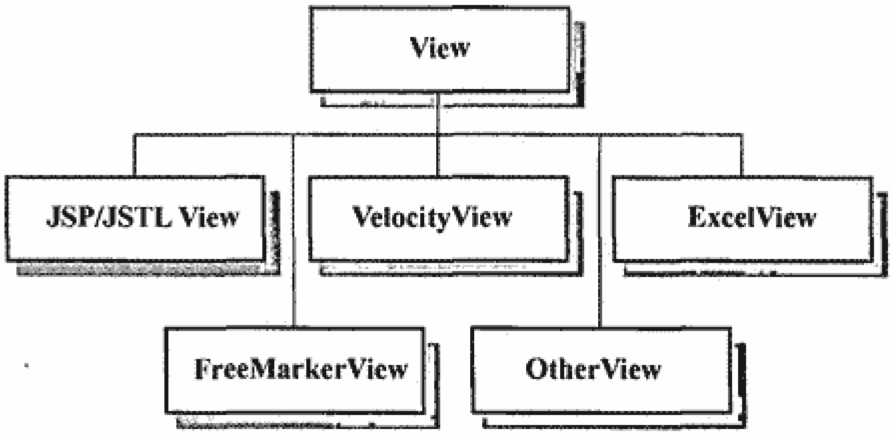

图23-2 视图抽象策略示意图

现在，视图模板与模型数据的合并逻辑，以及合并后的视图结果的输出逻辑，全部封装到了相应的View实现类中。DispatcherServlet只需要根据ModelAndView返回的信息，选择具体的View实现类做最终的具体工作即可。不过，与HandlerMapping帮助DispatcherServlet查找具体的Spring Controller以处理Web请求类似，DispatcherServlet现在需要依赖于某一个org.springframework.web.servlet.ViewResolver来帮它处理逻辑视图名与具体的View实例之间的映射对应关系。ViewResolver将根据ModelAndView中的逻辑视图名查找相应的View实现类，然后将查找的结果返回DispatcherServlet。DispatcherServlet最终会将ModelAndView中的模型数据交给返回的View来处理最终的视图渲染工作。

至此，整个DispatcherServlet的处理流程即告结束。

如果你还不足以从以上流程概略中分出哪个角色对应什么样的工作，图23-3所展示的Sequence图或许可以帮助大家加深一下印象。

在了解了Spring MVC中的基本概念之后，让我们开始着手开发我们的第一个基于Spring MVC的Web应用程序。

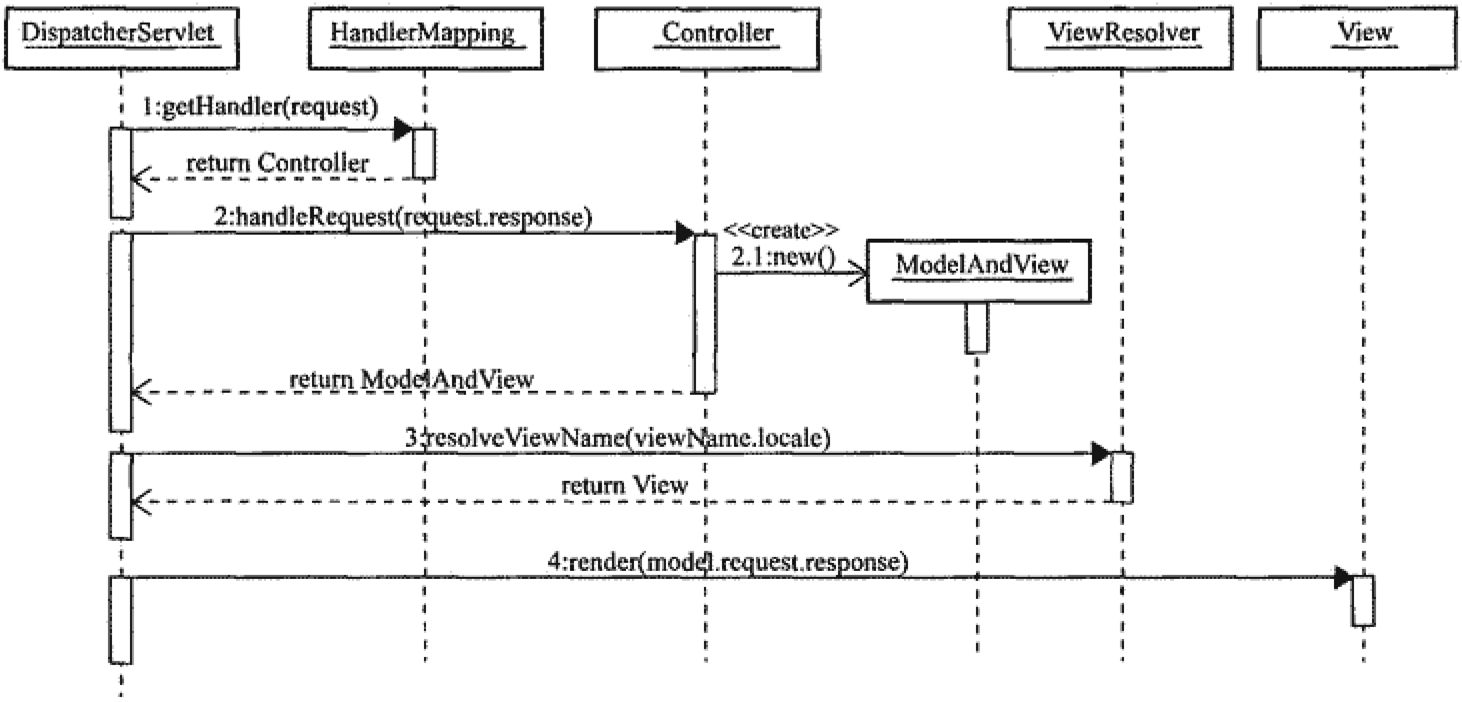

图23-3　Spring MVC中各角色交互图

23.2　实践出真知

如果你已经读到了这里，我完全能够理解你要实现第一个基于Spring MVC的Web应用程序的迫切心情，毕竟，一味地纸上谈兵令人腻烦，那么让我们开始一个我称之为`SimpleFX`的Web应用程序的开发吧！

一个外汇交易系统与其他的软件系统在本质上没有任何差别，为了能够便于一些数据的管理，我们也需要一个用于后台管理的应用程序。既然是讲解Spring MVC，我们打算以Web应用程序的形式来开发后台管理功能，这个用于后台管理的Web应用程序我们暂且就称其为`SimpleFX`。因为不打算完全实现它所有的功能，只要能够通过几个功能的开发阐明Spring MVC的相应特点即可。整个后台管理程序的显示布局如图23-4所示。

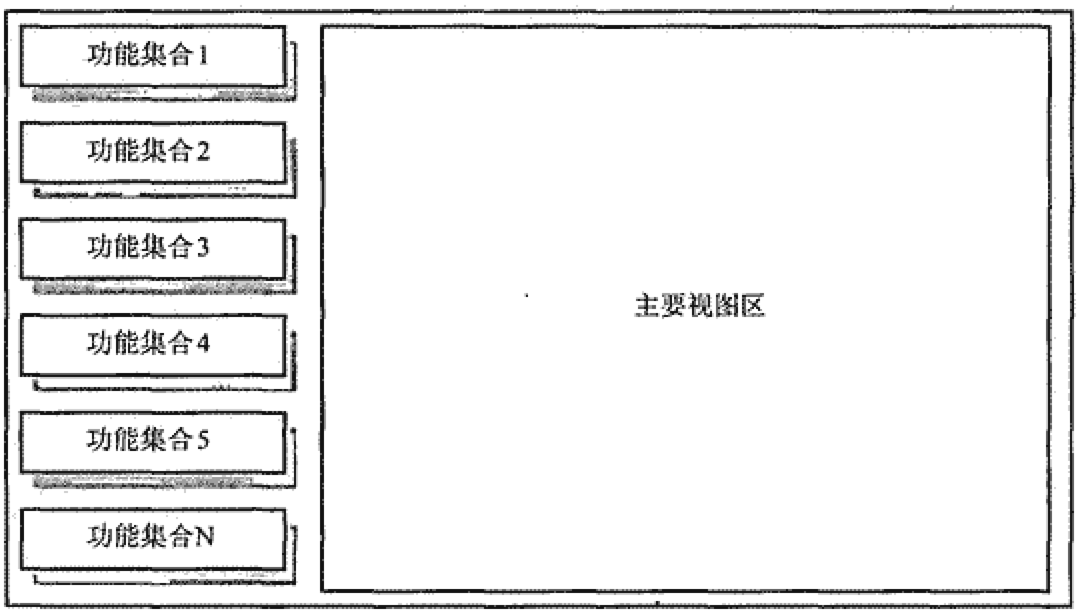

图23-4　后台管理程序显示布局示意图

其中，左边的各个FunctionSet表示相应的管理功能集合，我们可以通过点击这些FunctionSet的链接来触发相应的管理请求，剩下的部分为主画面显示，一看便知。

本次我们要开发的是一个简单的显示外汇交易当天评价汇率[①]的Web请求处理流程。在点击相应的链接之后，我们可以获取当天评价汇率的显示信息，类似于图23-5所示的主画面显示。

当日评价汇率

Currency Pair	TTM RATE
USD/JPY	121.53
EUR/USD	1.8950

图23-5　评价汇率显示主画面示意图

23.2.1　Spring MVC 应用的物理结构

虽然我们已经从总体上了解了Spring MVC框架在处理Web请求过程中所涉及的几个主要成员，但目前为止，它们只是逻辑上的概念，对于我们来说，只是一座座的空中楼阁。我们现在得面对现实，从实体结构上看一下，一个典型的基于Spring MVC的Web应用程序，通常是如何组装而成的（听起来好像我们在造一辆汽车的感觉），然后再真正地开始着手构建我们的第一个基于Spring MVC的Web应用程序。

一个基于Spring MVC框架的Web应用，依然是一个遵循Servlet规范的Web应用程序，自然，它也就拥有一个遵循Servlet规范的Web应用程序应有的目录结构和相应的部署描述符（Deployment Descriptor）文件。只不过，在此基础上，基于Spring MVC框架的Web应用会再增加两个（或者多个）配置文件，整个图景如图23-6所示。

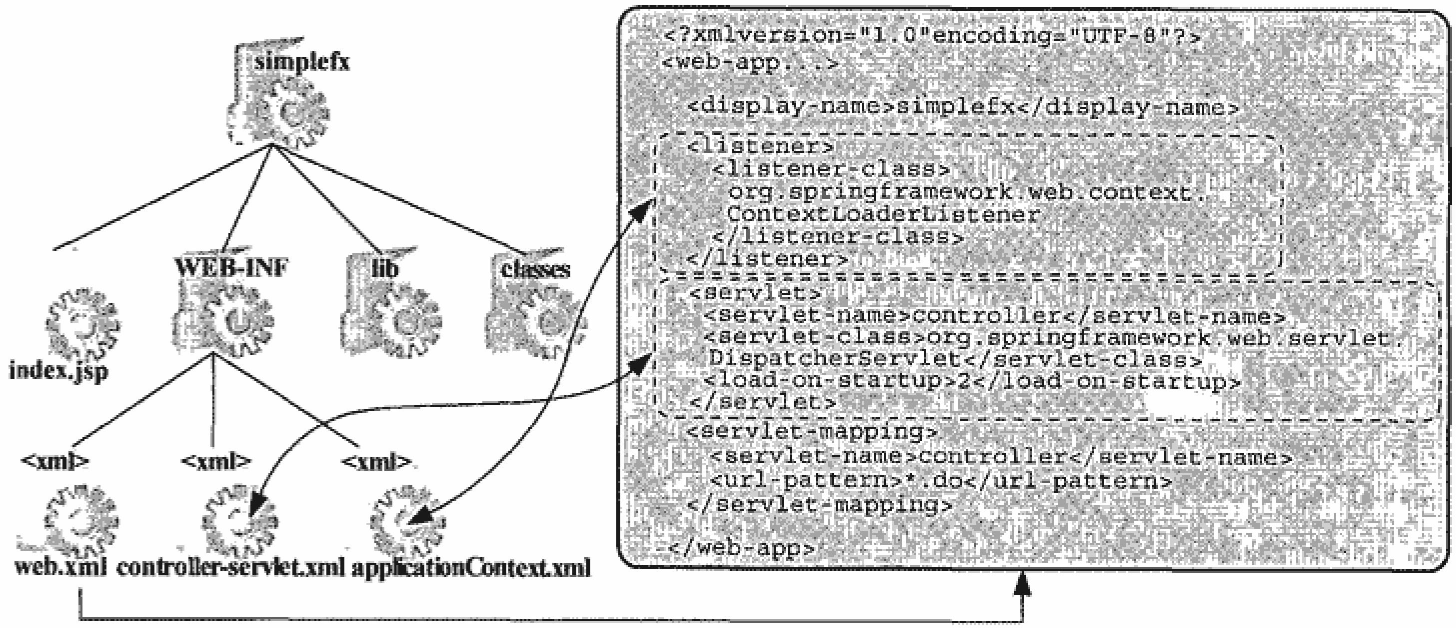

图23-6　Spring MVC配置结构示意图

① 外汇交易市场分为两类，一个是银行间市场，一个是客户市场，客户市场会以银行间市场每天9点55分发布的银行间汇率作为TTM(中间值)，然后在TTM汇率的基础上发送相应的交易汇率给客户进行交易，外汇系统中通常都会保存每一天的TTM汇率，我们在这里为了简化流程，避免参数的传递，只检索当天的TTM汇率。

我们以web.xml作为出发点，它是整个Web应用程序的部署描述符文件，这是所有基于Servlet规范的Web应用程序都有的东西，我想大家对它应该并不陌生。在图23-6右边的就是web.xml的具体内容。当然，在这个部署描述符文件中，我们还可以根据需要添加其他元素，比如可能需要进行字符编码的转换，那会增加一个Filter的定义，或者添加welcome文件列表以及异常处理页面相关内容等。不过为了避免混淆视听，我们这里仅挑了与Spring MVC有关的元素进行重点阐述。

1. ContextLoaderListener与/WEB-INF/applicationContext.xml

从web.xml开始，我们首先通过<listener>元素增加了一个ServletContextListener的定义，即org.springframework.Web.context.ContextLoaderListener。ContextLoaderListener的职责在于，它将为整个的Web应用程序加载顶层的WebApplicationContext（ROOT WebApplicationContext）。该顶层WebApplicationContext主要用于提供应用所使用的中间层服务。我们所使用的数据源（DataSource）定义、数据访问对象（DAO）定义、服务对象（Services）定义等，都在该WebApplicationContext中注册。你完全可以将其比作独立运行的应用程序中我们所使用的ClassPathXmlApplicationContext或者FileSystemXmlApplicationContext。只不过，WebApplicationContext专门用于Web环境下，在这种容器中，我们可以使用在讲解Spring的IoC容器部分提到的自定义scope来注册相应的bean定义了，包括request、session等。

ContextLoaderListener加载的WebApplicationContext的默认配置文件路径为/WEB-INF/applicationContext.xml，这也就是图23-6中的/WEB-INF/目录下存在一个applicationContext.xml配置文件的原因。该文件符合通用的Spring IoC容器配置文件格式。实际开发中，不管是出于团队并行开发效率的考虑，还是出于便于管理的因素考虑，很少会使用默认的单一的/WEB-INF/applicationContext.xml来管理整个Web应用程序的所有中间层服务对象。我们要么会按照应用程序的层次进行配置文件的分割，要么会按照系统功能模块进行配置文件的分割。当存在多个分割后的配置文件的时候，ContextLoaderListener的默认加载行为将成为我们的制约。这时，我们可以通过在web.xml中指定名称为contextConfigLocation的配置参数来打破默认行为的制约，如代码清单23-3所示。

代码清单23-3　contextConfigLocation配置参数使用代码示例

```
<context-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>/WEB-INF/applicationContext.xml,/WEB-INF/applicationContext-
      module1.xml</param-value>
</context-param>

<listener>
      <listener-class>org.springframework.Web.context.ContextLoaderListener
      </listener-class>
</listener>
```

我们可以在<param-value>中通过逗号或者空格来分割多个配置文件路径，甚至使用ANT类型的路径表达式。更多信息可以参阅org.springframework.web.context.ContextLoader类的Javadoc文档，因为ContextLoaderListener最终委派该类进行WebApplicationContext的加载。

ContextLoaderListener之外的选择

如果你不得不继续使用Servlet 2.2的Web容器，或者那些并不支持ServletContextListener特性的Servlet 2.3容器，那么不得不放弃ContextLoaderListener，转而使用org.springframework.

Web.context.ContextLoaderServlet加载顶层的WebApplicationContext。

ContextLoaderServlet完成与ContextLoaderListener相同的工作。需要注意的是，你得通过调整load-on-startup的值，让它在当前Web应用程序中使用的其他Servlet之前启动。稍后将讲述，通过ContextLoaderServlet或者ContextLoaderListener加载的顶层WebApplicationContext并非只为自己而生。

你可以通过如下的方式配置ContextLoaderServlet让其为你服务:

```
<servlet>
    <servlet-name>contextLoader</servlet-name>
    <servlet-class>org.springframework.Web.context.ContextLoaderServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
```

ContextLoaderListener或者ContextLoaderServlet加载相应路径下的容器配置文件，并在构建完成相应的顶层WebApplicationContext后，将该顶层WebApplicationContext绑定到ServletContext。如果想获取绑定到ServletContext的WebApplicationContext，我们可以通过org.springframework.web.context.support.WebApplicationContextUtils类。这样就不需要知道顶层WebApplicationContext绑定到ServletContext的时候使用的Key是什么，例如：


```
WebApplicationContext wac = ➥
WebApplicationContextUtils.getWebApplicationContext(getServletContext());
或者
WebApplicationContext wac = ➥
WebApplicationContextUtils.getRequiredWebApplicationContext(getServletContext());
wac.getBean("serviceBeanName");
```

了解这一点的主要目的在于，绑定到ServletContext的顶层WebApplicationContext并非只能在基于Spring MVC的Web应用程序中使用。通过ContextLoaderListener或者ContextLoaderServlet将其绑定到ServletContext，任何类型的Web应用程序（当然指的是基于Java平台技术的Web应用程序），只要能够获取ServletContext的引用，就能获取并使用该WebApplicationContext。换句话说就是，即使当前使用的Web开发框架没有类似于Spring提供的基于IoC容器的中间层业务对象管理支持，通过在web.xml中注册ContextLoaderListener或者ContextLoaderServlet，也可以获取这种支持。所谓的集成Spring与第三方Web应用框架，最多只是在此基础上做进一步的扩展。

2. DispatcherServlet与XXX-servlet.xml

对于web.xml中org.springframework.Web.servlet.DispatcherServlet的注册，表面看起来并无太多玄机，与注册其他Servlet并无二致，但事实并非如此。我们说过，DispatcherServlet是基于Spring MVC框架Web应用程序的Front Controller，它将负责几乎所有对应当前Web应用程序的Web请求的处理。如果我们之前使用过Struts框架的话，那么可以将DispatcherServlet比做Struts框架中的ActionServlet，实际上二者所完成的工作，性质上应该是一致的。Struts框架中的ActionServlet会从外部的struts-config.xml配置文件中读取Web处理过程中所需要的各种数据，包括form bean定义、action以及action mapping等。与之类似，DispatcherServlet也使用了一个外部化的配置文件，用来配置Spring MVC框架在处理Web请求过程中所涉及的各个组件，包括HandlerMapping定义、Controller定义、ViewResolver定义等。该外部化的配置文件存在的默认路径也是/WEB-INF/，名称需要参照web.xml中定义的DispatcherServlet的<servlet-name>来决

定。比如，我们当前定义的DispatcherServlet对应的<servlet-name>为**controller**，那么，默认的配置文件即对应/WEB-INF/controller-servlet.xml，也就是说，DispatcherServlet对应的默认配置文件名称，将在<servlet-name>的值的基础上后缀-servlet.xml。如果我们定义DispatcherServlet的<servlet-name>为simplefx，那么它的配置文件名称就应该为simplefx-servlet.xml，以此类推。

<servlet-name>-servlet.xml与/WEB-INF/applicationContext.xml相同，也符合Spring IoC容器配置文件格式。只不过，相对于/WEB-INF/applicationContext.xml来说，<servlet-name>servlet.xml有自己的职责。<servlet-name>-servlet.xml主要负责配置基于Spring MVC框架的Web应用程序所使用的各种Web层组件，具体内容看起来如代码清单23-4所示。

代码清单23-4　<servlet-name>-servlet.xml配置代码示例

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http:// www.springframework.org/schema/beans" ➥
xmlns:xsi="http:// www.w3.org/2001/XMLSchema-instance" ➥
xmlns:p="http:// www.springframework.org/schema/p" ➥
xmlns:context="http:// www.springframework.org/schema/context" ➥
xsi:schemaLocation="http:// www.springframework.org/schema/beans ➥
http:// www.springframework.org/schema/beans/spring-beans-2.5.xsd ➥
http:// www.springframework.org/schema/context ➥
http:// www.springframework.org/schema/context/spring-context-2.5.xsd">
        ...
    <bean id="handlerMapping" ➥
    class="org.springframework.Web.servlet.handler.BeanNameUrlHandlerMapping">
    </bean>

    <bean id="viewResolver" ➥
    class="org.springframework.Web.servlet.view.InternalResourceViewResolver">
      <property name="prefix" value="/WEB-INF/jsp/"/>
       <property name="suffix" value=".jsp"/>
    </bean>

    <bean name="/infoList.do" class="..InfoListController">
    </bean>
...
</beans>
```

DispatcherServlet启动之后将加载<servlet-name>-servlet.xml配置文件，并构建相应的WebApplicationContext。该WebApplicationContext将之前通过ContextLoaderListener加载的顶层WebApplicationContext（ROOT WebApplicationContext）作为父容器（Parent ApplicationContext）。这样，如果需要，<servlet-name>-servlet.xml中注册的各种面向Web层的组件，也可以注入来自顶层WebApplicationContext的依赖了。所以，我们也就不难想象出<servlet-name>-servlet.xml对应的WebApplicationContext和默认的/WEB-INF/applicationContext.xml对应的顶层WebApplicationContext之间的逻辑依赖关系了（见图23-7）。

即使Web层通常很薄，但随着Web应用程序的开发进程，单一的<servlet-name>-servlet.xml文件有可能成为团队并行开发的瓶颈，或者其身材也可能会越加臃肿。出于这些因素考虑，我们需要寻求分割单一的<servlet-name>-servlet.xml文件的解决方法，而DispatcherServlet的contextConfigLocation初始化参数就是为此而生的。代码清单23-5给出了该初始化参数的使用配置代码示例。

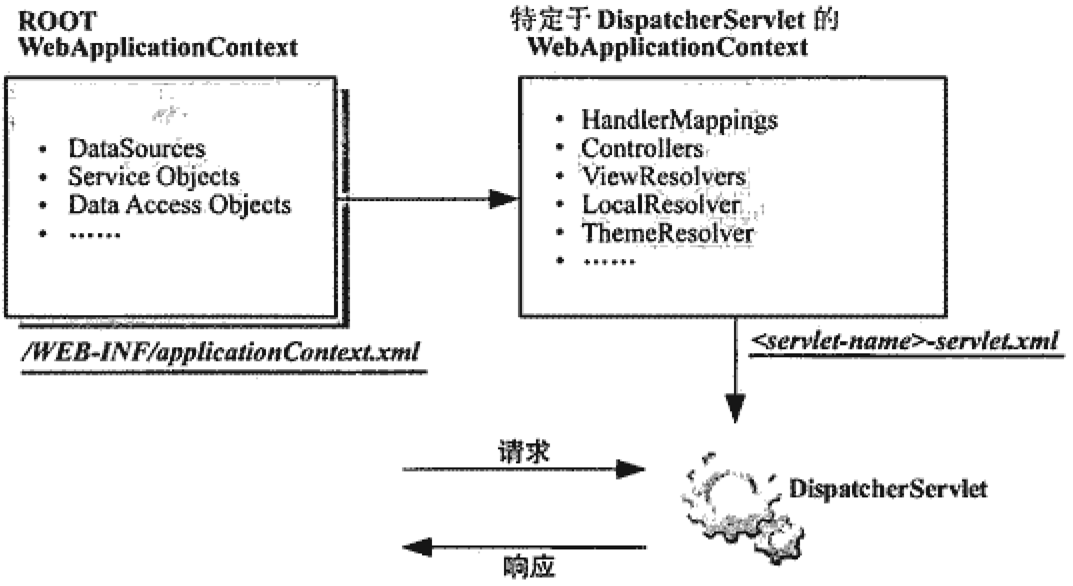

图23-7 顶层（ROOT）的WAC和特定于DispatcherServlet的WAC之间的逻辑依赖关系

代码清单23-5 DispatcherServlet的contextConfigLocation初始化参数配置代码示例

```
<servlet>
    <servlet-name>controller</servlet-name>
    <servlet-class>org.springframework.Web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/controller-servlet.xml,/WEB-INF/ ➥
        module1-servlet.xml,..</param-value>
    </init-param>
    <load-on-startup>2</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>controller</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

我们可以指定多个配置文件，用逗号或空格来分隔。更多定制DispatcherServlet的信息，可以参照Spring参考文档或者相应类的Javadoc。

现在，你已经对基于Spring MVC的Web应用程序都由哪些物理实体组成，以及如何定制它们有了一定的认识，那么，热身活动结束！

23.2.2 按部就班地开始工作

我们要实现的当日评价汇率显示流程可以简单描述，如图23-8所示。

从浏览器中点击后台管理画面的相应链接之后，Web请求将被发送到DispatcherServlet进行处理。DispatcherServlet将寻求相应的HandlerMapping对Web请求进行分析，然后调用匹配结果对应的Controller实现，具体到当前场景就是我们要实现的TTMRateListController。TTMRateListController处理完毕将视图名称ttmRateList和模型数据一同返回，然后DispatcherServlet则借助于相应的ViewResolver，根据返回的视图名选择相应的视图（ttmRateList.jsp）并显示，这就是整个流程。下面则是实现这个Web处理流程的详细步骤，各位上眼瞧啦！

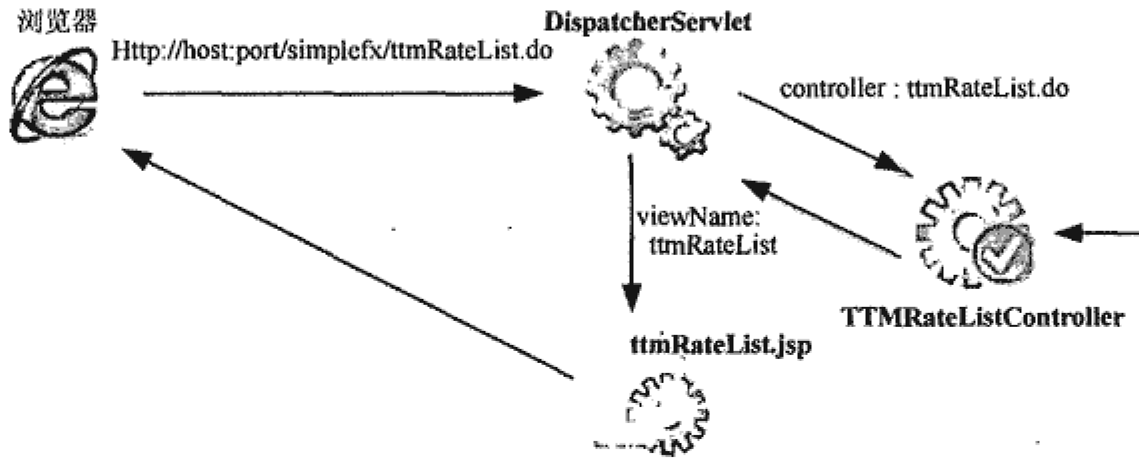

图23-8　当日评价汇率Web处理流程演示

当日评价汇率Web处理流程实现步骤概况如下。

(1) **配置基础装备**。每个符合Java EE规范的Web应用程序都需要符合相应的目录结构，如图23-6所示。工作之初，我们需要构建Web应用的基础结构。不过，现在的IDE通常都有良好的Web开发支持，使用Eclipse IDE的话，添加WTP（Web Tools Platform）或者MyEclipse插件的支持即可。剩下的只是按照Wizard的说明构建一个Web应用的工程就行了，我们这里要建立的是`simplefx`工程。`simplefx`工程建立之后，我们还有如下两件事情要做。

a) **配置web.xml**。我们需要将`org.springframework.Web.servlet.DispatcherServlet`和`org.springframework.Web.context.ContextLoaderListener`通过`<servlet>`和`<listener>`元素添加到web.xml部署描述符文件中，原因我们之前已经领教了。另外，出于其他目的考虑，我们还可以添加相应的`Filter`以及`ServletContextListener`以处理字符编码和Log4j初始化等配置内容，这些完全根据当前应用程序的需求情况来决定。现在，我们的web.xml看起来如代码清单23-6所示。

代码清单23-6　`simplefx`对应的web.xml配置文件内容摘录

```
<Web-app... >
    <display-name>simplefx</display-name>

    <filter>
    <filter-name>encodingFilter</filter-name>
    <filter-class>org.springframework.Web.filter.CharacterEncodingFilter</filter-class>
        <init-param>
            <param-name>encoding</param-name>
            <param-value>UTF-8</param-value>
        </init-param>
    </filter>
    <filter-mapping>
        <filter-name>encodingFilter</filter-name>
        <servlet-name>controller</servlet-name>
    </filter-mapping>

    <listener>
    <listener-class>org.springframework.Web.util.Log4jConfigListener</listener-class>
    </listener>
    <listener>
    <listener-class>org.springframework.Web.context.ContextLoaderListener</listener-class>
    </listener>

    <servlet>
```

```
        <servlet-name>controller</servlet-name>
    <servlet-class>org.springframework.Web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>2</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>controller</servlet-name>
        <url-pattern>*.do</url-pattern>
    </servlet-mapping>

    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
</Web-app>
```

其他配置元素可能随着内容的深入继续添加。

b) **WebApplicationContext文件的添加。**我们需要在/WEB-INF/目录下添加org.springframework.Web.servlet.DispatcherServlet和org.springframework.Web.context.ContextLoaderListener对应的WebApplicationContext配置文件（/WEB-INF/applicationContext.xml和/WEB-INF/[servletname]-servlet.xml）。这完全可以从Spring下载文件包中的sample目录下任何一个Web应用程序中复制过来。只不过，需要根据命名规则修改相应的文件名称。当然，复制过来的文件内容最好是清空，只保留可以在当前应用中能够通用的配置内容。对于我们的simplefx来说，我们暂且清空对应的两个WebApplicationContext配置文件内容。

以上基础设施构建完成之后，我们开始“盖楼”。

(2) **开发独立的业务逻辑。**对于一个设计良好的Web应用程序来说，虽然Web层依赖于业务层对象，但业务层却不应该对Web层有任何的依赖。Web层只应该看作是公开业务逻辑的一种视角或者交互方式。这样做或者说这样看待系统的好处在于，业务层可以独立设计并实现，而不需要关心最终通过什么手段将服务公开给用户。鉴于这样的理念，我们完全可以从业务层开始着手设计和实现我们的当日评价汇率显示逻辑，而且在稍后介绍Spring Remoting支持的时候，你将会更深地感受到这种看待系统的视角所带给我们的系统设计与实现的福利。

既然涉及评价汇率，那么首先需要设计并实现的对象即对应评价汇率的域对象（Domain Object），其简单定义如代码清单23-7所示。

代码清单23-7 TTMRate对象定义

```
public class TTMRate implements Serializable
{
    private static final long serialVersionUID = 2641189625594925925L;

    private TradeDate  frontDate;
    private String     currencyPair;
    private BigDecimal value;

    public TTMRate(TradeDate  frontDate,String currencyPair,BigDecimal value)
    {
        this.frontDate = frontDate;
        this.currencyPair = currencyPair;
        this.value = value;
    }

    public TradeDate getFrontDate() {
        return frontDate;
```

```
    }
    public void setFrontDate(TradeDate frontDate) {
        this.frontDate = frontDate;
    }
    public String getCurrencyPair() {
        return currencyPair;
    }
    public void setCurrencyPair(String currencyPair) {
        this.currencyPair = currencyPair;
    }
    public BigDecimal getValue() {
        return value;
    }
    public void setValue(BigDecimal value) {
        this.value = value;
    }
    // 其他方法定义，比如toString()等
}
```

每个营业日内对应系统中可交易的货币对，都存在一个评价汇率，所以，TTMRate中定义了必需的三个字段分别对应营业日、货币对和对应的评价汇率值。

为了能够使各种客户端（当然包括Web客户端）能够获取当日的评价汇率，我们需要规定相应的接口并公开给相应的客户端使用。现在这个接口定义很简单，定义如下：

```
public interface ITTMRateService {
    List<TTMRate> getTTMRatesToday();
}
```

只要客户端调用getTTMRatesToday()方法，就将返回当前营业日内，系统所支持的所有货币对对应的评价汇率。

单有接口并不能工作，我们还需要给出相应的实现。正常情况下，我们需要结合相应的数据访问对象构建一个ITTMRateService的实现，以便能够从系统数据库中获取真正生产环境下的数据。不过，我不打算这么做。出于简化实例考虑，我们现在只实现一个ITTMRateService的Mock对象。真实环境下的ITTMRateService实现，如果你感兴趣可以自己把玩一下，ITTMRateService的Mock对象定义如代码清单23-8所示。

代码清单23-8　ITTMRateService的Mock对象定义代码示例

```
public class MockTTMRateService implements ITTMRateService
{
      public List<TTMRate> getTTMRatesToday()
      {
             TradeDate tradeDate20080302 = TradeDate.valueOf("20080302");
             TTMRate USD_JPY = new TTMRate(tradeDate20080302,"USD/JPY",➥
             new BigDecimal("121.53"));
             TTMRate EUR_USD = new TTMRate(tradeDate20080302,"EUR/USD",➥
             new BigDecimal("1.8950"));
             List<TTMRate> rateList = new ArrayList<TTMRate>();
             rateList.add(USD_JPY);
             rateList.add(EUR_USD);

             return rateList;
      }
}
```

我们假设当天营业日对应20080302，并将对应该日期的USD/JPY和EUR/USD两货币对的

评价汇率信息添加到返回列表中。整个Web应用的中间层服务支持，默认都是通过/WEB-INF/applicationContext.xml注册（即ROOT WebApplicationContext）。所以，现在我们要将ITTMRateService的实现增加到Web应用程序顶层容器的配置文件中，如代码清单23-9所示。

代码清单23-9 增加ITTMRateService实现类到顶层WebApplicationContext的配置示例

```
<beans xmlns="http:// www.springframework.org/schema/beans" ➥
xmlns:xsi="http:// www.w3.org/2001/XMLSchema-instance" ➥
xmlns:p="http:// www.springframework.org/schema/p" ➥
xmlns:aop="http:// www.springframework.org/schema/aop" ➥
xmlns:context="http:// www.springframework.org/schema/context" ➥
xmlns:jee="http:// www.springframework.org/schema/jee" ➥
xmlns:tx="http:// www.springframework.org/schema/tx" ➥
xsi:schemaLocation=" ➥
http:// www.springframework.org/schema/aop ➥
http:// www.springframework.org/schema/aop/spring-aop-2.5.xsd ➥
http:// www.springframework.org/schema/beans ➥
http:// www.springframework.org/schema/beans/spring-beans-2.5.xsd ➥
http:// www.springframework.org/schema/context ➥
http:// www.springframework.org/schema/context/spring-context-2.5.xsd ➥
http:// www.springframework.org/schema/jee ➥
http:// www.springframework.org/schema/jee/spring-jee-2.5.xsd ➥
http:// www.springframework.org/schema/tx ➥
http:// www.springframework.org/schema/tx/spring-tx-2.5.xsd">

     <bean id="ttmRateService"➥
     class="cn.spring21.simplefx.service.MockTTMRateService">
     </bean>

 </beans>
```

真实环境下，我们可能还需要向该容器配置文件中添加数据源（DataSource）等一系列依赖对象。在此之后，我们才真正开始Web层的开发。

(3) **添加Web请求入口。**要让用户能够通过Web方式访问我们的评价汇率服务，我们得为用户提供一个入口点。首先在后台管理的主画面左侧的FunctionSet中添加超链接，如下所示：

```
<a href="<c:url value='ttmRateList.do'/>">当日评价汇率</a>
```

在用户点击该链接之后，浏览器将以http:// host:port/simplefx/ttmRateList.do的形式向服务器发起Web请求。我们在web.xml中将所有以.do结尾的请求模式都映射给了DispatcherServlet来处理。接下来的事情就由DispatcherServlet接手了。

(4) **添加HandlerMapping。**DispatcherServlet在接收到Web请求之后，将寻求相应的HandlerMapping进行Web请求到具体的Controller实现的匹配。所以，我们需要为DispatcherServlet提供一个HandlerMapping的实现。Spring MVC框架默认提供了多个HandlerMapping的实现。我们将在稍后详细介绍。不过，当前我们暂且使用org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping，它将根据URL与Controller的bean定义的名称进行匹配。现在我们将它添加到DispatcherServlet特定的WebApplicationContext中，即/WEB-INF/controller-servlet.xml，如代码清单23-10所示。

代码清单23-10 BeanNameUrlHandlerMapping配置相关代码示例

```
<beans xmlns="http:// www.springframework.org/schema/beans" ➥
xmlns:xsi="http:// www.w3.org/2001/XMLSchema-instance" ➥
```

```
xmlns:p="http:// www.springframework.org/schema/p" ➥
xmlns:context="http:// www.springframework.org/schema/context" ➥
xsi:schemaLocation="http:// www.springframework.org/schema/beans ➥
http:// www.springframework.org/schema/beans/spring-beans-2.5.xsd ➥
http:// www.springframework.org/schema/context ➥
http:// www.springframework.org/schema/context/spring-context-2.5.xsd">

    <bean id="handlerMapping" ➥
    class="org.springframework.Web.servlet.handler.BeanNameUrlHandlerMapping">
    </bean>

</beans>
```

实际上，如果没有配置任何HandlerMapping的话，Spring MVC也会默认使用BeanNameUrlHandlerMapping进行URL到具体Controller的匹配。所以，以上的bean定义并非必须的。但是，这样做的目的是要明确告知你，DispatcherServlet所使用的HandlerMapping要在哪里进行配置。稍后你还将看到，如果默认的HandlerMapping的匹配模式并不讨人喜欢，我们还可以替换掉它，转而使用自己喜欢的HandlerMapping实现。

总之，我们使用BeanNameUrlHandlerMapping进行URL到具体Controller的匹配之后，BeanNameUrlHandlerMapping将根据http:// host:port/simplefx/ttmRateList.do这一URL信息，在当前容器内寻找名称为/ttmRateList.do的Controller定义。这将是我们下一步要实现的东西。

(5) 实现对应的Controller并添加到配置文件。 针对当前评价汇率的Web请求，我们需要实现一个Controller来处理它。通常情况下，我们会扩展Spring MVC的org.springframework.Web.servlet.mvc.AbstractController来实现具体的Controller。正如我们的处理流程图（图23-8）所演示的那样，我们的Controller名称为TTMRateListController，具体定义如代码清单23-11所示。

代码清单23-11　TTMRateListController类定义代码示例

```
public class TTMRateListController extends AbstractController{
    private ITTMRateService ttmRateService;
    private String viewName;

    @Override
    protected ModelAndView handleRequestInternal(HttpServletRequest ➥
    request,HttpServletResponse response) throws Exception {
        List<TTMRate> ttmRateList = getTtmRateService().getTTMRatesToday();

        ModelAndView mav = new ModelAndView(getViewName());
        mav.addObject("ttmRates", ttmRateList);
        return mav;
    }

    public ITTMRateService getTtmRateService() {
        return ttmRateService;
    }

    public void setTtmRateService(ITTMRateService ttmRateService) {
        this.ttmRateService = ttmRateService;
    }

    public String getViewName() {
        return viewName;
    }
```

```
    public void setViewName(String viewName) {
        this.viewName = viewName;
    }
}
```

在TTMRateListController中，我们引用了ITTMRateService并借助它获取了当日的评价汇率列表。viewName是将要转向的逻辑视图名称，可以将它写死到代码。不过，通过WebApplicationContext的依赖注入，我们可以获取更多的灵活度。在视图名称和要显示的模型数据全都获取之后，我们构造了一个ModelAndView来封装这两部分信息，并返回。

只是实现了TTMRateListController，DispatcherServlet并不会知道它的存在，也不会知道如何引用到它，所以，我们还得将TTMRateListController添加到DispatcherServlet特定的WebApplicationContext中。现在，/WEB-INF/controller-servlet.xml看起来更加充实了，如代码清单23-12所示。

代码清单23-12 controller-servlet.xml配置内容代码示例


```
<beans xmlns="http://www.springframework.org/schema/beans" ➥
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ➥
xmlns:p="http://www.springframework.org/schema/p" ➥
xmlns:context="http://www.springframework.org/schema/context" ➥
xsi:schemaLocation="http://www.springframework.org/schema/beans ➥
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd ➥
http://www.springframework.org/schema/context ➥
http://www.springframework.org/schema/context/spring-context-2.5.xsd">

<bean id="handlerMapping" ➥
class="org.springframework.Web.servlet.handler.BeanNameUrlHandlerMapping">
</bean>

<bean name="/ttmRateList.do" class="..TTMRateListController">
    <property name="ttmRateService" ref="ttmRateService"/>
    <property name="viewName" value="ttmRateList"/>
</bean>

</beans>
```

这里需要注意的是，TTMRateListController的ttmRateService注入的依赖来自org.springframework.Web.context.ContextLoaderListener加载的顶层WebApplicationContext。我们说过，DispatcherServlet特定的WebApplicationContext以顶层WebApplicationContext作为父容器。另外，BeanNameUrlHandlerMapping要求相应的Controller的bean定义名称以/开头，所以，我们得使用<bean>的name属性来标志Controller对应的bean定义。

(6) 添加**ViewResover**。TTMRateListController通过ModelAndView返回了一个逻辑视图名ttmRateList，而DispatcherServlet要根据这个逻辑视图名查找相应的视图实现，就需要一个ViewResolver来帮它。目前我们还不能在DispatcherServlet的特定WebApplicationContext中找到一个ViewResolver的实现，所以至少得添加一个，否则也太令DispatcherServlet失望了。

Spring MVC也为ViewResolver提供了多种实现，我们当前决定使用比较普及的JST/JSTL作为视图技术，所以，将org.springframework.Web.servlet.view.InternalResourceViewResolver添加到了DispatcherServlet特定的WebApplicationContext中，如代码清单23-13所示。

代码清单23-13 添加ViewResolver后的配置文件内容示例

```
<beans xmlns="http:// www.springframework.org/schema/beans" ➥
```

```
xmlns:xsi="http:// www.w3.org/2001/XMLSchema-instance" ➥
xmlns:p="http:// www.springframework.org/schema/p" ➥
xmlns:context="http:// www.springframework.org/schema/context" ➥
xsi:schemaLocation="http:// www.springframework.org/schema/beans ➥
http:// www.springframework.org/schema/beans/spring-beans-2.5.xsd ➥
http:// www.springframework.org/schema/context ➥
http:// www.springframework.org/schema/context/spring-context-2.5.xsd">

    <bean id="handlerMapping" ➥
    class="org.springframework.Web.servlet.handler.BeanNameUrlHandlerMapping">
    </bean>

    <bean name="/ttmRateList.do" class="..TTMRateListController">
    <property name="ttmRateService" ref="ttmRateService"/>
    <property name="viewName" value="ttmRateList"/
    </bean>

    <bean id="viewResolver" ➥
    class="org.springframework.Web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/jsp/"/>
        <property name="suffix" value=".jsp"/>
    </bean>
</beans>
```

我们为InternalResourceViewResolver指定了prefix和suffix两个属性的自定义值。这样，当我们的逻辑视图名（viewName）为ttmRateList的时候，InternalResourceViewResolver将去寻找[prefix]+viewName+[suffix]名称的视图模板文件，也就是/WEB-INF/jsp/ttmRateList.jsp。不过看来该JSP文件还没有创建。

(7) **实现相应视图。**我们得提供InternalResourceViewResolver根据逻辑视图名所查找的视图模板文件/WEB-INF/jsp/ttmRateList.jsp。我们通过相应的IDE在/WEB-INF/jsp/目录下创建一个ttmRateList.jsp文件如代码清单23-14所示。

代码清单23-14 ttmRateList.jsp视图模板文件代码示例

```
<%@ page language="java" contentType="text/html; charset=UTF-8" ➥
pageEncoding="UTF-8"%>
<%@ taglib uri="http:// java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE html PUBLIC "-// W3C// DTD HTML 4.01 Transitional// EN" ➥
"http:// www.w3.org/TR/html4/loose.dtd">
    <html>
    <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Insert title here</title>
    </head>
    <body>
    <table width="517" border="1" align="center" cellpadding="0" cellspacing="0" ➥
    bordercolor="#33FFFF">
        <CAPTION>当日评价汇率</CAPTION>
        <tr bgcolor="#999900"><td width="250"><div align="center">Currency ➥
        Pair</div></td><td width="261"><div align="center">TTM RATE</div></td>
        </tr>
        <c:forEach items="${ttmRates}" var="ttmRate">
            <tr>
                <td><div align="center">${ttmRate.currencyPair}</div></td>
```

```
            <td><div align="center">${ttmRate.value}</div></td>
        </tr>
    </c:forEach>
</table>
</body>
</html>
```

我们使用JSTL对模板中的显示逻辑进行管理，将在TTMRateListController中添加到Model-AndView的模型数据取出来并进行显示，也可以使用自己熟悉的自定义Taglib，比如Struts的自定义Taglib，或者Spring MVC提供的自定义Taglib，但绝不要使用Scriptlet来做这个工作，原因我们之前已经说过了。

提示 对于这种表格数据的现实，你也可以使用DisplayTag或者eXtrmeTable等开源的Taglib，这将大大减少我们的重复工作，提高工作效率。

在我们完成以上工作并将simplefx部署到Tomcat或者其他的Web Container甚至应用服务器之后，我们将见到之初所预期的画面。很是令人振奋，不是吗？

23.3 小结

在深入Spring MVC腹地之前，为了帮助你尽快融入到Spring MVC的世界中，我们首先从总体上对Spring MVC的设计和实现流程进行了简单的介绍。实例通常是快速了解一件事物的最好方式，所以，我们在简单分析了基于Spring MVC的Web应用程序物理结构之后，一起开发了我们的实例应用simplefx。接下来，我们将在simplefx的基础上，进一步扩展视野，引入Spring MVC中的概念。

第24章 近距离接触Spring MVC主要角色

本章内容

- 忙碌的协调人HandlerMapping
- 我们的亲密伙伴Controller
- ModelAndView
- 视图定位器ViewResolver
- 各司其职的View

要恭喜你的是，即使不再继续余下的旅程，你也已经能够处理Spring MVC Web应用开发过程中80%的场景，但却可能不得不重复处理Web开发过程中的某些方面，比如Locale的设置、数据的绑定和验证、form页面的显示和提交等。为了能够避免每个处理流程都需要实现这些东西，Spring MVC提供了更细粒度的组件支持，而这也将带给我们更加舒适的开发体验。虽然我们同样可以实现这么一套组件体系，但已经有先行者替我们做了这部分工作，岂不是更好？

如果你想进一步避免Web开发过程中的某些重复工作，如果你想"坐享其成"（好像不是个好词儿哦），如果你想了解Spring MVC框架的更多奥秘，让我们继续出发！

24.1 忙碌的协调人 HandlerMapping

HandlerMapping帮助DispatcherServlet进行Web请求的URL到具体处理类的匹配。之所以称为HandlerMapping是因为，在Spring MVC中，并不只局限于使用org.springframework.web.servlet.mvc.Controller作为DispatcherServlet的次级控制器来进行具体的Web请求的处理。实际上，在稍后介绍HandlerAdaptor的时候，你就会了解到，我们也可以使用其他类型的次级控制器，包括Spring MVC提供的除了Controller之外的次级控制器类型，或者第三方Web开发框架中的Page Controller组件（如Struts的Action），而所有这些次级控制器类型，在Spring MVC中都称作Handler，我想这就是HandlerMapping这一名称的由来了。HandlerMapping要处理的也就是Web请求到相应Handler之间的映射关系。如果你接触过Struts框架的话，可以将HandlerMapping与Struts框架的ActionMapping概念进行类比。只不过HandlerMapping的职责更加明确，使用上也更加灵活。

HandlerMapping的定义很简单，因为大部分实际工作都是由其相应的子类来完成的，其定义如下所示：

```
public interface HandlerMapping {
    String PATH_WITHIN_HANDLER_MAPPING_ATTRIBUTE = HandlerMapping.class.getName() +➥
```

```
        ".pathWithinHandlerMapping";
    HandlerExecutionChain getHandler(HttpServletRequest request) throws Exception;
}
```

刚看到该接口的定义你或许会说我糊弄人，“HandlerMapping返回给DispacherServlet的，哪里是一个Controller啊？明明是一个HandlerExecutionChain嘛”。各位毋需动怒，HandlerMapping接口定义的确如此，不过我所言也非虚啊，只不过是为了简化概念而已。我想，各位不想还没了解Spring MVC大体是什么东西的时候，就吃上一顿“概念大餐”吧？即使现在，我也不打算带大家详细了解这个返回的HandlerExecutionChain到底是个什么东西。目前为止，只需要大家知道，HandlerExecutionChain中确实包含了用于处理具体Web请求的Handler，仅此而已。

24.1.1 可用的 HandlerMapping

Spring MVC默认提供了多个HandlerMapping的实现供我们选用，这当然包括我们在最初所接触的BeanNameUrlHandlerMapping，除此之外，还有如下几个。

- **SimpleUrlHandlerMapping**。较之BeanNameUrlHandlerMapping，该实现类进一步解除了请求URL与Handler的beanName之间的耦合，并且支持更灵活的映射表达方式。
- **ControllerClassNameHandlerMapping**。我们将在第27章中接触到它。
- **DefaultAnnotationHandlerMapping**。Spring 2.5之后的Spring MVC引入了基于注解的配置方式，我们将在第26章中详细了解它。

我们已经提前接触过BeanNameUrlHandlerMapping了。下面我们详细看一下SimpleUrlHandlerMapping的特性与使用方法。

SimpleUrlHandlerMapping

使用BeanNameUrlHandlerMapping进行Web请求到具体Handler的映射管理，需要我们保证视图模板中的请求路径，必须与容器中对应的Handler的beanName一致，如图24-1所示。

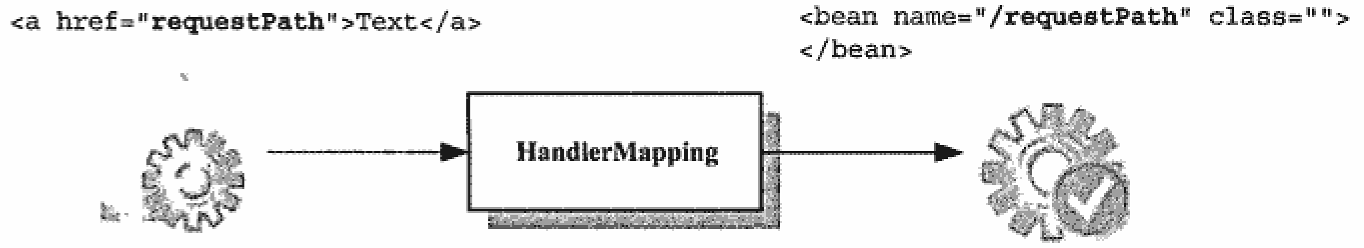

图24-1 BeanNameUrlHandlerMapping的映射处理示意图

也就是说，BeanNameUrlHandlerMapping只做了一个传话人，并没有对请求路径与Handler之间的映射做太多的关注（有点儿不动脑子的意思）。从好的方面说，遵循Convention Over Configuration的理念可以减少配置量，提高开发效率。从坏的方面说，BeanNameUrlHandlerMapping强制我们的Handler的bean定义名称，必须去匹配视图中的链接路径。如果表现层和Web层分别由不同的人员开发，并且人员之间没有合适的交流渠道，那么就有些令双方人员勉为其难了。

SimpleUrlHandlerMapping比BeanNameUrlHandlerMapping做的工作要多一些，可以说尽到了自己作为中间人的那份职责。它允许视图一方和handler一方自由活动，只不过最后由SimpleUrlHandlerMapping进行“统筹”。如果我们将处理当日评价汇率流程中使用的BeanNameUrlHandlerMapping替换成SimpleUrlHandlerMapping，DispatcherServlet特定的WebApplicationContext中的内容看起来如代码清单24-1所示。

代码清单24-1　SimpleUrlHandlerMapping使用配置示例

```
<beans xmlns="http:// www.springframework.org/schema/beans" xmlns:xsi="http:// ➥
www.w3.org/2001/XMLSchema-instance" ➥
xmlns:p="http:// www.springframework.org/schema/p" xmlns:context="http:// ➥
www.springframework.org/schema/context" ➥
xsi:schemaLocation="http:// www.springframework.org/schema/beans http:// ➥
www.springframework.org/schema/beans/spring-beans-2.5.xsd ➥
http:// www.springframework.org/schema/context http:// ➥
www.springframework.org/schema/context/spring-context-2.5.xsd">

        <bean id="handlerMapping" ➥
        class="org.springframework.Web.servlet.handler.SimpleUrlHandlerMapping">
            <property name="mappings">
            <props>
                <prop key="ttmRateList.do">ttmRateListController</prop>
                ...
            </props>
            </property>
        </bean>

        <bean name="ttmRateListController" class="..TTMRateListController">
            <property name="ttmRateService" ref="ttmRateService"></property>
            <property name="viewName" value="ttmRateList"></property>
        </bean>

        <bean id="viewResolver" ➥
        class="org.springframework.Web.servlet.view.InternalResourceViewResolver">
            <property name="prefix" value="/WEB-INF/jsp/"/>
            <property name="suffix" value=".jsp"/>
        </bean>
</beans>
```

现在，TTMRateListController可以起任何名字，视图中的链接也可以独立提供，只需要通过SimpleUrlHandlerMapping明确指定一下二者的对应关系就行。SimpleUrlHandlerMapping不对二者做任何限制，如图24-2所示。

使用SimpleUrlHandlerMapping的另一个好处就是，我们可以使用类似于ANT路径形式的模式匹配，这样我们就可以通过各种表达式，将一组或者多组拥有某种相似特征的Web处理请求映射给相应的Handler处理，类似于代码清单24-2所示的样子。

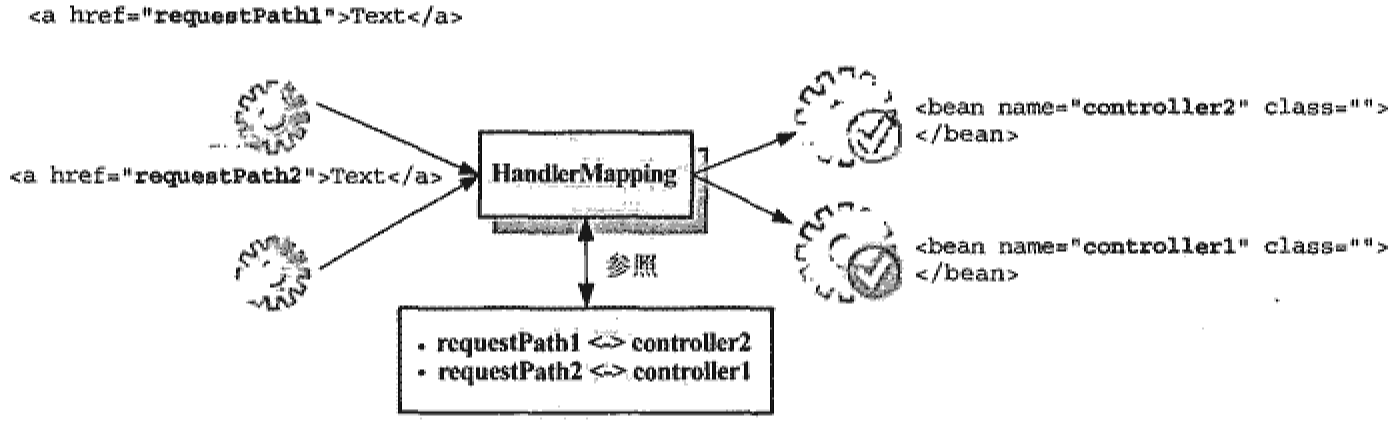

图24-2　SimpleUrlHandlerMapping映射管理示意图

代码清单24-2 SimpleUrlHandlerMapping的另一个使用配置代码的示例

```
<bean id="handlerMapping"➥
class="org.springframework.Web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="mappings">
            <value>
                ttmRateList.do=ttmRateListController
                /**/*list.do=genericListingController
                /user*.do=userManagerController
                /module/*.do=ModuleInOneController
            </value>
        </property>
</bean>
```

当然，说了这些不是说SimpleUrlHandlerMapping要比BeanNameUrlHandlerMapping有多好。我们更注重的是，在开发的过程中，可以根据情况，灵活地选用相应的HandlerMapping来使用。

注意 什么？你发现同样是SimpleUrlHandlerMapping的mappings属性，我们用了<props>和<value>两种不同的元素进行配置，你不知道到底哪一个对？是不是写错了？难道你忘了Spring提供的org.springframework.beans.propertyeditors. PropertiesEditor是用来干什么的了吗？

24

24.1.2 HandlerMapping 执行序列（Chain Of HandlerMapping）

在基于Spring MVC的Web应用程序中，我们可以为DispatcherServlet提供多个HandlerMapping供其使用。DispatcherServlet在选用HandlerMapping的过程中，将根据我们所指定的一系列HandlerMapping的优先级进行排序，然后优先使用优先级在前的HandlerMapping。如果当前的HandlerMapping能够返回可用的Handler，DispatcherServlet则使用当前返回的Handler进行Web请求的处理，而不再继续询问其他的HandlerMapping。否则，DispatcherServlet将继续按照各个HandlerMapping的优先级进行询问，直到获取一个可用的Handler为止。

HandlerMapping的优先级规定遵循Spring框架内一贯的Ordered接口所规定的语义。Spring MVC中可用的HandlerMapping实现全都实现了Ordered接口。假设我们优先使用SimpleUrlHandlerMapping进行Handler的映射管理，其次使用BeanNameUrlHandlerMapping，那么就可以在DispatcherServlet特定的WebApplicationContext中增加如代码清单24-3所示的配置内容。

代码清单24-3 HandlerMapping指定优先级配置代码示例

```
<bean id="handlerMapping" class="org.springframework.Web.servlet.handler.➥
SimpleUrlHandlerMapping">
    <property name="order" value="1"/>
    <property name="mappings">
        <value>
            ...
        </value>
    </property>
</bean>

<bean id="defaultHandlerMapping" ➥
class="org.springframework.Web.servlet.handler.BeanNameUrlHandlerMapping">
</bean>
```

如果不为HandlerMapping明确指定order，那么默认值为Integer.MAX_VALUE，对应最低优先级。所以，拥有order值为1的SimpleUrlHandlerMapping较之BeanNameUrlHandlerMapping优先被调用。

24.2 我们的亲密伙伴 Controller

Controller是Spring MVC框架支持的用于处理具体Web请求的handler类型之一。在使用Spring MVC框架开发Web应用程序过程中，它是我们接触最多的角色，几乎每个请求流程的实现都需要与Controller打交道（否则具体的处理逻辑写到哪里去呢？），所以称其为“亲密伙伴”也不为过吧！

要实现一个具体的Controller，我们当然可以直接实现Controller接口，其定义如下：

```
public interface Controller {
    ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse ➥
    response) throws Exception;
}
```

但更多时候，我们可能会寻求使用Spring MVC提供的更细粒度的Controller框架类。直接实现Controller接口当然没有问题，这让我们可以随心所欲地实现Web处理过程中的所有关注点，但这通常需要我们关注更多底层的细节，比如请求参数的抽取、请求编码的设定、国际化信息的处理、Session数据的管理等。而实际上，可能这些细节或者关注点是所有的Controller都需要的。我们就应该想办法让这些通用的逻辑可以以某种方式被复用。这也就是Spring MVC提供了一套Controller实现体系的原因。它帮助我们更好地处理了Web请求处理过程中的某些通用关注点。图24-3给出了Spring MVC中Controller的继承层次体系的概况。

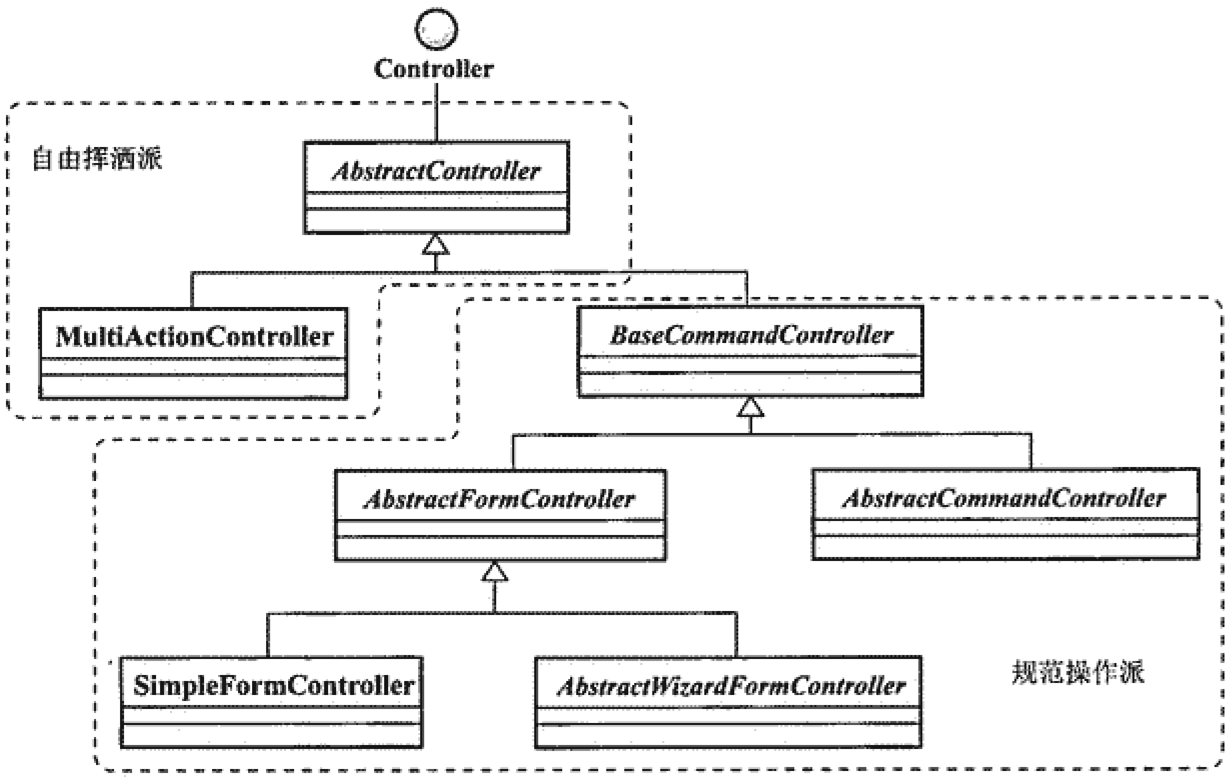

图24-3　Controller层次体系

为了便于理解，我们不妨把Controller划分为如下两类。

- **自由挥洒派的Controller**。之所以将AbstractController和MultiActionController归为

自由挥洒一派，是因为基本上只要你熟悉Servlet API，就可以随意处理Web请求了。如果你是从Servlet时代走过来的，那么会发现这一派的做事风格让你觉得很亲切，从HttpServletRequest中获取参数，然后验证，调用业务层逻辑，最终返回一个ModelAndView，甚至，你都可以直接通过HttpServletResponse输出最终的视图。不过自由倒是自由了，杂七杂八要处理的细节也很多，所以，使用AbstractController和MultiActionController之前最好先考察一下当前场景是否合适。如果要处理的底层细节过多，那么不妨求助一下“规范操作派”的各位仁兄！

- **规范操作派的Controller。** 以BaseCommandController为首的规范操作一派，对Web处理过程中的某些通用逻辑做了进一步的规范化封装处理，规范化的方面主要包括：
  - 自动抽取请求参数并绑定到指定的Command对象①；
  - 提供了统一的数据验证方式，BaseCommandController及其子类可以接收一组org.springframework.validation.Validator以进行数据验证，我们可以根据具体数据提供相应的Validator实现；
  - 规范化了表单（Form）请求的处理流程，并且对简单的多页面表单请求处理提供支持。

虽然这套“制度”对我们的自由挥洒进行了限定，但只要我们熟悉了这套规则，依然可以游刃于其间。

各门派我们已经介绍完了，下面是各派中实力比较雄厚的诸位弟子的表演时间……

24

24.2.1 AbstractController

AbstractController是整个Controller继承层次的起源（多少有点儿少林正宗弟子的意味），该类通过模板方法模式（Template Method Pattern）帮我们解决了如下几个通用关注点：

- 管理当前Controller所支持的请求方法类型（GET/POST）；
- 管理页面的缓存设置，即是否允许浏览器缓存当前页面；
- 管理执行流程在会话（Session）上的同步。

而我们所要做的，只不过是在AbstractController所公开的handleRequestInternal(request,response)模板方法中实现具体Web请求处理过程中的其他逻辑。

> **注意** 有关AbstractController所处理的通用关注点的设置参数，可以参考Spring文档或者AbstractController类的Javadoc。

虽然我们已经扩展AbstractController实现了一个TTMRateListController，但多少让人感觉过于简单了。为了进一步向你展示AbstractController的无尽魅力，我们不妨对TTMRateListController进行一下扩展。

> TTMRateListController只获取当日评价汇率。为了一般化评价汇率显示功能，我们希望能够检索指定营业日的评价汇率。当用户没有明确指定营业日的时候，返回当日评价汇率，否则返回指定营业日对应的评价汇率。而这也正是大多生产环境下Web应用的需求形式。

---

① Spring MVC中的Command对象类似于Struts的FormBean的概念，只不过并不强制实现或者扩展规定的类型定义，几乎任何类型的对象都可以，但最好是JavaBean，这样Spring MVC可以充分使用PropertyEditor机制进行数据绑定过程中的类型转换。

为了达到以上扩展要求，我们需要对TTMRateListController及其相关组件进行一系列的重构工作。

重构评价汇率请求处理流程的步骤如下所示。

(1) 重构业务层接口及实现。为了能够提供根据营业日进行检索的功能，我们需要一般化ITTMRateService的服务方法。重构后的ITTMRateService接口定义如下所示：

```
public interface ITTMRateService {
    List<TTMRate> getTTMRatesByTradeDate(TradeDate tradeDate);
}
```

相应的实现类现在也需要提供进一步的支持，我们依然先采用MockTTMRateService提供模拟数据。现在MockTTMRateService定义如代码清单24-4所示。

代码清单24-4　重构后的MockTTMRateService定义

```
public class MockTTMRateService implements ITTMRateService{
private Map<TradeDate,List> mockData = new HashMap<TradeDate,List>();

public MockTTMRateService(){
    TradeDate tradeDate20080302 = TradeDate.valueOf("20080302");
    TTMRate USD_JPY = new TTMRate(tradeDate20080302,"USD/JPY",new BigDecimal("121.53"));
    TTMRate EUR_USD = new TTMRate(tradeDate20080302,"EUR/USD",new BigDecimal("1.8950"));
    List<TTMRate> ratesOf20080302 = new ArrayList<TTMRate>();
    ratesOf20080302.add(USD_JPY);
    ratesOf20080302.add(EUR_USD);
    mockData.put(tradeDate20080302, ratesOf20080302);
    // ……添加更多模拟数据
    // mockData.put(TradeDate.today(),...);
}

public List<TTMRate> getTTMRatesByTradeDate(TradeDate tradeDate) {
    List<TTMRate> ttmRateList = mockData.get(tradeDate);
    if(ttmRateList == null)
        return new ArrayList<TTMRate>();
    return ttmRateList;
    }
}
```

当然，只有出于测试目的才像我们这样构造数据。实际系统中通常需要从数据库中抽取相应的评价汇率。但不管怎么样，我们暂且有了一个可用的ITTMRateService实现了。

既然早先我们已经将MockTTMRateService添加到了顶层WebApplicationContext中，现在可以免去这一步，直接进入Web层的重构工作。

(2) 重构TTMRateListController。TTMRateListController现在要接收并处理Web请求的查询参数，然后根据参数调用相应服务对象，最终返回处理结果，其具体实现逻辑如代码清单24-5所示。

代码清单24-5　重构后的TTMRateListController定义

```
public class TTMRateListController extends AbstractController{
   private ITTMRateService ttmRateService;
   private String viewName;

   @Override
   protected ModelAndView handleRequestInternal(HttpServletRequest ➥
   request,HttpServletResponse response) throws Exception {
```

```
        ModelAndView mav = new ModelAndView(getViewName());

        // 比[request.getParameter("tradeDate")]更好的选择
        String tradeDateStr = ServletRequestUtils.getStringParameter(request, "tradeDate");
        TradeDate tradeDate = TradeDate.today();

        // 可以根据情况在这里添加验证逻辑
        if(StringUtils.isNotEmpty(tradeDateStr)){
            tradeDate = TradeDate.valueOf(tradeDateStr);
            mav.addObject("tradeDate", tradeDateStr);
        }

        List<TTMRate> ttmRateList = getTtmRateService().getTTMRatesByTradeDate(tradeDate);
        mav.addObject("ttmRates", ttmRateList);
        return mav;
    }
    // getter和setter方法定义等
    // ...
}
```

实际上，我们还是对TTMRateListController处理的流程做了简化。实际情况可能是，当检查到获取的营业日字符串不合法的时候，需要将相应的错误信息也添加到ModelAndView，然后在视图中显示。

(3) **修改视图模板**。现在我们需要添加一个表单，以便用户可以输入要查询的营业日。修改后的JSP模板文件如代码清单24-6所示。

代码清单24-6 修改后的JSP试图模板代码示例

```
<%@ page language="java" contentType="text/html; charset=UTF-8" ➥
pageEncoding="UTF-8"%>
<%@ taglib uri="http:// java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE html PUBLIC "-// W3C// DTD HTML 4.01 Transitional// EN" ➥
"http:// www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
<form name="" method="post" action="<c:url value='ttmRateList.do'/>">
    <label>营业日
    <input type="text" name="tradeDate" value="${tradeDate}">
    </label>
    <label>
    <input type="submit" name="Submit" value="检索">
    </label>
</form>
<table width="517" border="1" cellpadding="0" cellspacing="0" bordercolor="#33FFFF">
    <CAPTION>当日评价汇率</CAPTION>
    <tr bgcolor="#999900"><td width="250"><div align="center">Currency ➥
    Pair</div></td><td width="261"><div align="center">TTM RATE</div></td>
    </tr>
    <c:forEach items="${ttmRates}" var="ttmRate">
        <tr>
            <td><div align="center">${ttmRate.currencyPair}</div></td>
```

```
            <td><div align="center">${ttmRate.value}</div></td>
        </tr>
    </c:forEach>
</table>
</body>
</html>
```

相对最初的示例代码来说，我们只是多增加了一个表单的HTML定义。其中<input>所传递的参数名称tradeDate，恰好对应TTMRateListController获取的参数名称，这个是最底层的契约。

整个示例要用于实际生产环境还存在需要改进的地方，比如，更规范的数据合法性验证、错误流程的追加以及视图中错误信息的显示，甚至于视图的美化等。但我们已经通过它构建了通向更加完善的处理流程的基础。不管怎么说，现在已经能够看到重构后的初步成果了，见图24-4。

图24-4　重构后的画面显示

正如我们所看到的，虽然直接继承AbstractController实现Web处理逻辑没有太多限制，但往往需要关注太多的细节，比如参数抽取、数据验证之类。所以，对于简单的Web请求，可以通过这种方式来处理，如果涉及表单相关的Web请求，最好是使用BaseCommandController一族的相应子类进行扩展，如稍后将介绍的SimpleFormController。

24.2.2 MultiActionController

MultiActionController一旦出马，就身兼数职。对于一组逻辑上相近的Web请求来说，比如针对同一对象的CRUD（Create-Read-Update-Delete）操作，或者针对同一对象甚至多个对象的一组查询操作，我们可以将这些Web请求交给MultiActionController来统一处理，而不用分别为每个Web请求单独实现一个继承AbstractController的处理类。MultiActionController类似于Struts 1.1框架的DispatchAction，但比DispatchAction要复杂并且灵活得多。

MultiActionController继承了AbstractController，所以也就拥有了AbstractController所处理的那些通用关注点的能力。除此之外，MultiActionController还提供了以下功能。

- **请求参数到Command对象的绑定**。这看起来多少有点儿多管BaseCommandController闲事的意思。因为正规上说，BaseCommandController才是提供数据绑定功能的基类。不过，BaseCommandController提供的数据绑定和验证更多地服务于其子类的流程控制，而MultiActionController提供的数据绑定能力则比较独立，有点儿你爱用不用的意思。
- **通过Validator的数据验证**。这个功能与MultiActionController的数据绑定是一起的，一旦启用了到Command对象的数据绑定，那么，绑定过程中MultiActionController会自动调用我们注册的一系列Validator进行数据验证。当然，即使不用数据绑定功能，也同样可以自行决定是否调用相应的Validator。
- **细化的异常处理方法**。可以定义一系列的异常处理方法，用于处理Web请求处理过程中所抛出的特定类型的异常。

为了在MultiActionController中处理多个Web请求，我们需要定义多个Web请求处理方法，分

别对应每个Web请求的处理。这些Web请求处理方法可以定义在MultiActionController的子类中，也可以定义在某一个将来可以指定给MultiActionController的委派对象（delegate）内，但Web请求处理方法的签名必须符合一定的要求，如下所示：

```
(ModelAndView | Map | void) methodName(HttpServletRequest request, HttpServletResponse ➥
response[,(HttpSession session|Object command)]);
```

Web请求处理方法的名称可以取任何有意义的名字，但前两个方法参数是必须的，第三个参数是可选的，可以是HttpSession类型也可以是Object类型。如果是Object类型，则表明对应的是Command对象，那么MultiActionController就会帮我们绑定数据并执行数据验证了。方法的返回值有三种类型，分别对应如下语义。

- 返回ModelAndView表示正常的Web处理方法，后继的ViewResolver和View处理流程，依照之前的DispatcherServlet的流程进行。
- 返回Map表明只返回了模型数据，而没有返回逻辑视图名。这时，将寻求默认的视图名。这个工作由org.springframework.Web.servlet.RequestToViewNameTranslator负责，该类将在第27章中详细介绍。这里只需要知道，它能按照某种规则提供一个默认的逻辑视图名。
- 返回void，则表明既没有返回模型数据，也没有返回逻辑视图名。这时，我们认为，当前Web请求处理方法自行处理掉了视图的渲染和输出。
- 另外，Spring 2.5中可以返回String，代表逻辑视图名，没有相关的模型数据。

有了以上的规则限定，通常的MultiActionController定义看起来就如代码清单24-7所示的样子了。

代码清单24-7 MultiActionController实现类代码示例

```
public class GenericCRUDMultiActionController extends MultiActionController {
    public Map list(HttpServletRequest request, HttpServletResponse response){
        ...
    }

    public ModelAndView delete(HttpServletRequest request, HttpServletResponse response){
        ...
    }

    public ModelAndView update(HttpServletRequest request, HttpServletResponse ➥
    response, Object command){
        ...
    }
    ...
}
```

剩下我们所要做的就是，在每个方法内实现当前Controller要完成的功能。

不过，我们肯定有一个问题，如何让MultiActionController知道具体哪个Web请求进来之后，将由哪个方法来处理它？答案是MethodNameResolver。

1. MultiActionController的助理MethodNameResolver

org.springframework.Web.servlet.mvc.multiaction.MethodNameResolver的主要工作，就是帮助MultiActionController决定当前Web请求应该交给哪个方法来处理，其定义如下：

```
public interface MethodNameResolver {
    String getHandlerMethodName(HttpServletRequest request) throws ➥
```

```
        NoSuchRequestHandlingMethodException;
    }
```

有了MethodNameResolver这一助手，MultiActionController现在只需要询问一声当前需要调用哪个方法即可，如下代码所示：

```
protected ModelAndView handleRequestInternal(HttpServletRequest request, ➥
HttpServletResponse response) throws Exception {
    try {
        String methodName=this.methodNameResolver.getHandlerMethodName(request);
        return invokeNamedMethod(methodName, request, response);
    }
catch (NoSuchRequestHandlingMethodException ex) {
    return handleNoSuchRequestHandlingMethod(ex, request, response);
}
}
```

MethodNameResolver为MultiActionController提供了灵活的Web请求到对应处理方法的映射策略，包括根据Web请求的URL进行映射，或者根据某个参数值进行映射等。在考虑提供自定义的MethodNameResolver之前，我们有必要先了解一下Spring MVC都提供了哪几个可用的MethodNameResolver实现，以免误入“重新发明轮子”之地。

MethodNameResolver本身作为一个策略接口定义，在Spring MVC框架内默认提供了如下三种策略实现。

InternalPathMethodNameResolver。如果没有为MultiActionController明确指定任何MethodNameResolver，那么InternalPathMethodNameResolver将作为默认的MethodNameResolver实现，以进行Web请求与具体处理方法间的映射解析。InternalPathMethodNameResolver将提取URL最后一个（/）之后的部分并去除扩展名，作为要返回的方法名称，比如：

- /simplefx/macontroller/listing.do
- /simplefx/macontroller/create.do
- /simplefx/macontroller/update.do
- /simplefx/macontroller/delete.do

如果将/simplefx/macontroller/*通过HandlerMapping指定映射到某个MultiActionController实现来处理，那么，对应以上4组URL的Web请求，将分别由这个MultiActionController的listing、create、update以及delete4个方法来处理。

当然，我们也可以通过设定InternalPathMethodNameResolver的prefix和suffix来有限的修饰映射行为，比如：

```
<bean id="internalPathMethodNameResolver" ➥
class="org.springframework.Web.servlet.mvc.multiaction.InternalPathMethodNameResolver">
    <property name="prefix" value="rate_"/>
</bean>
```

当使用如上的InternalPathMethodNameResolver定义来映射刚才4组URL的时候，对应的处理方法名应该变成rate_listing、rate_create、rate_update以及rate_delete。

> **注意**　除非需要指定prefix或者suffix这样的额外属性，作为默认MethodNameResolver实现的InternalPathMethodNameResolver，并不需要明确添加到容器中。

PropertiesMethodNameResolver。PropertiesMethodNameResolver与InternalPathMethod-

NameResolver的唯一相同点在于，它们都是基于请求的URL进行映射（二者有共同的父类Abstract-UrlMethodNameResolver），但它比InternalPathMethodNameResolver灵活。如果从Handler-Mapping与MethodNameResolver都是处理映射这一点来看，InternalPathMethodNameResolver相当于BeanNameUrlHandlerMapping，而PropertiesMethodNameResolver则相当于SimpleUrl-HandlerMapping。

通过PropertiesMethodNameResolver可以指定完全匹配的映射关系，或者使用ANT形式的路径匹配模式所表达的映射关系，例如：

```
<bean id="propsMethodNameResolver" ➥
class="org.springframework.Web.servlet.mvc.multiaction.PropertiesMethodNameResolver">
    <property name="mappings">
        <value>
            /listing.do=listingRates
            /update.do=updateRate
            /create*.do=addNewRate
        </value>
    </property>
</bean>
```

MultiActionController中定义的方法不允许重载，所以，映射中只需要指定方法名即可，不需要任何参数信息。

24

ParameterMethodNameResolver。结合ParameterMethodNameResolver使用的MultiAction-Controller，应该是最贴近Struts 1.1的DispatchAction的行为了。不过，ParameterMethodName-Resolver赋予了MultiActionController比DispatchAction更灵活多变的请求到处理方法的映射方式。

ParameterMethodNameResolver允许我们根据请求中的某个参数的值作为映射的方法名，也允许我们使用请求中一组参数来映射处理方法名称。下面是这两种策略的详细情况。

(1) 根据请求中某个参数的值作为映射后的方法名。在Web请求提交之后，我们可以附带一个参数，专门指定由MultiActionController的哪个方法来处理当前请求，如图24-5所示。

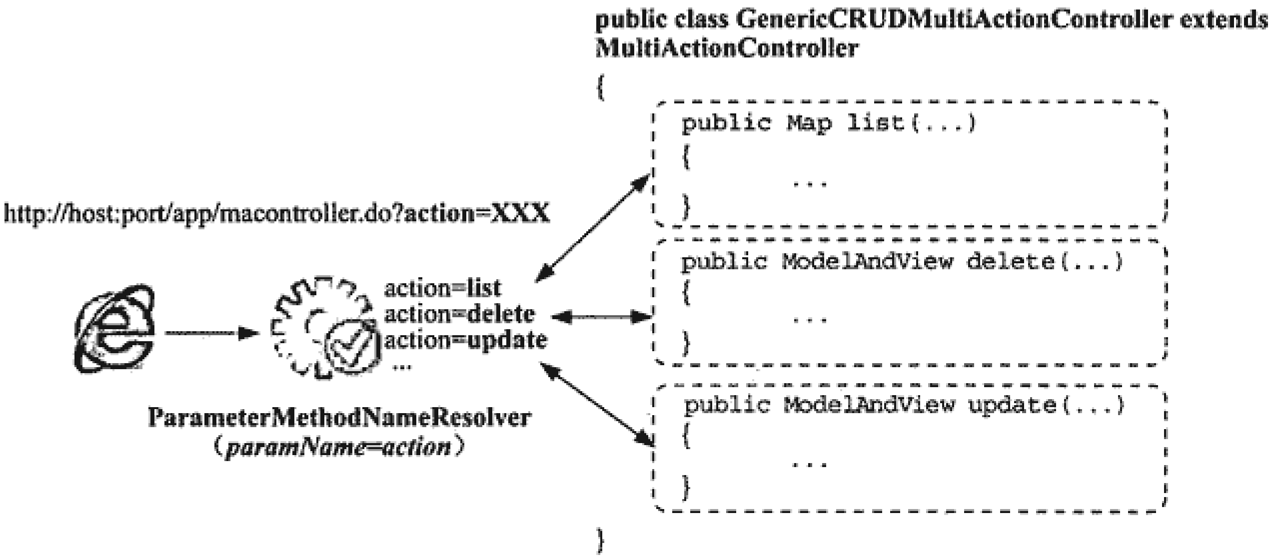

图24-5 paramName属性决定的映射示意图

ParameterMethodNameResolver默认检测的参数名称为action，恰好如图24-5所示。我们也可以通过ParameterMethodNameResolver的setParamName(String)方法来更改默认的参数名称。如果

如下所示，指定paramName为methodName的话：

```
<bean class="org.springframework.Web.servlet.mvc.multiaction.ParameterMethodNameResolver">
    <property name="paramName" value="methodName"/>
</bean>
```

那么，HTTP GET形式发送的URL看起来就类似于http:// host:port/simplefx/macontroller?methodName=list。当然，以HTTP GET形式演示可以让你有一个直观的认识。你同样可以通过HTTP POST的形式发送该参数（可能需要在客户端根据事件做些手脚，玩过HTML和Javascript的应该都知道）。

(2) 根据请求中的一组参数作为映射后的方法名。在某个页面中存在多种行为选择的时候，可以让每一种行为对应一个参数，这样，在Web请求提交之后，ParameterMethodNameResolver可以根据提交后的行为参数来调用相应的处理方法，如图24-6所示。

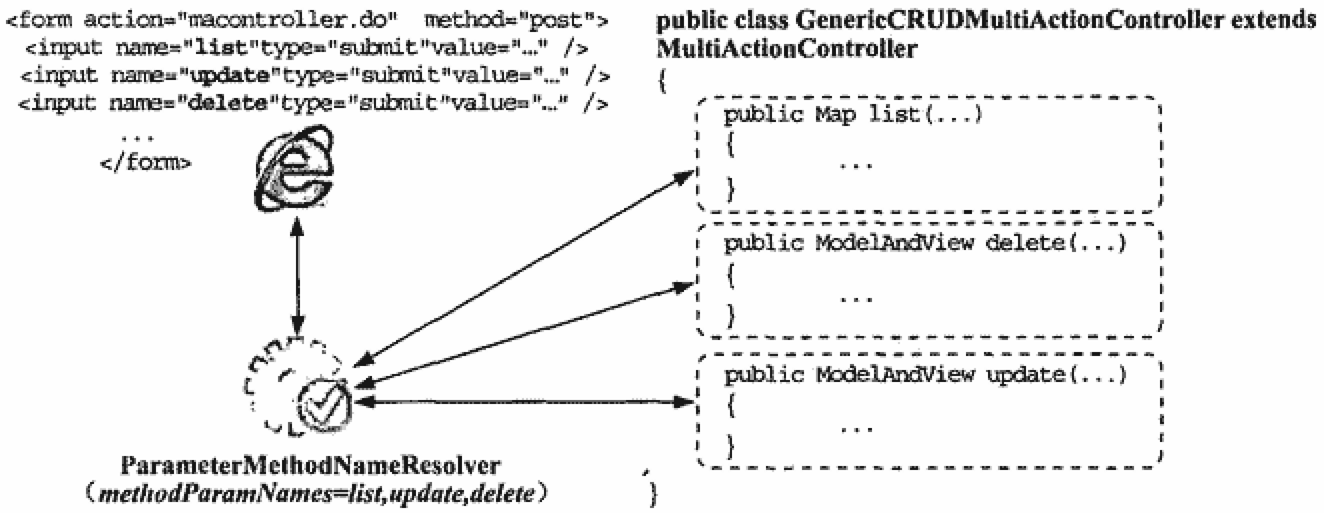

图24-6　methodParamNames属性决定的映射示意图

我们通过methodParamNames属性为ParameterMethodNameResolver指定一组要检测的参数名。ParameterMethodNameResolver将以指定的一组参数名作为基准，对Web请求中的参数进行检测。如果发现存在其中某个参数，则将当前Web请求映射到与参数相同名称的处理方法。假设图24-6中只为methodParamNames指定list和update两个值，那么，在名称为delete的submit按钮提交之后，ParameterMethodNameResolver将无视该参数的存在。这时就可能导致当前请求找不到相应的处理方法进行处理的问题。保险起见，最好通过ParameterMethodNameResolver的defaultMethodName属性指定一个默认处理方法，如下所示：

```
<bean class="org.springframework.Web.servlet.mvc.multiaction.ParameterMethodNameResolver">
    <property name="methodParamNames" value="list,update,delete"/>
    <property name="defaultMethodName" value="list"/>
</bean>
```

如果ParameterMethodNameResolver的methodParamNames和paramName两个属性都设置了的话，那意味着两种策略同时使用。这时，以methodParamNames属性为代表的映射策略将被优先考虑。

基本上，这三种MethodNameResolver就可以满足我们日程开发的需要了。如果实在没有合适的，那再去实现一个也不晚。要做的只不过是替换一下具体的MethodNameResolver策略实现类而已。这也就是MultiActionController比Struts 1.1的DispatchAction更具灵活性之所在了。

2. **MultiActionController应用演示**

因为评价汇率通常是不能够更新或者删除的，所以，为了演示MultiActionController的使用，我们得重新寻找一个场景。在SimpleFX的后台管理程序中，操作员属于不同的组（Group），每个组的管理权限不同，有的组可以读取或者更新管理信息，而有的组可能只能读取信息。我们可以增加这样的组，更新某个组的权限，甚至删除某个组。这些针对同一实体的操作请求，我们可以将它们映射到同一个MultiActionController进行处理。

我们对整个操作员组的处理场景进行了统一的流程建模，这包括稍后将使用SimpleFormController实现的处理流程部分，图24-7中是对整个流程的简要概括。

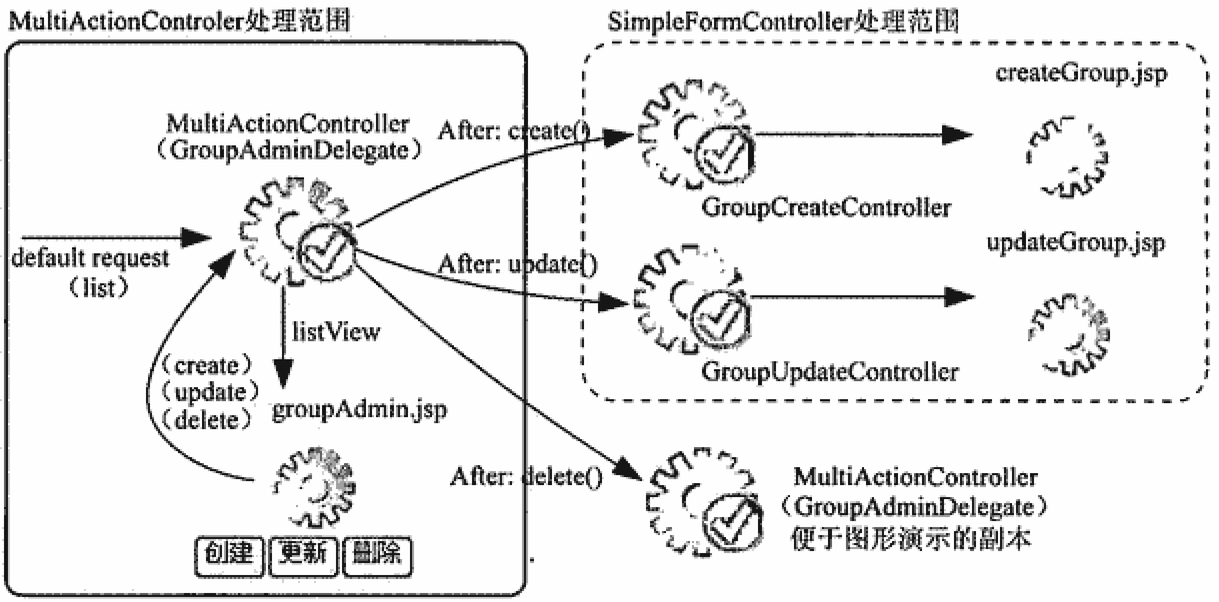

图24-7 操作员组权限管理实现流程示意图

当前，我们主要关心左边矩形框中的流程处理部分。在我们点击后台管理画面相应链接发起Web请求之后，我们将操作员组管理相关的Web请求全部交由MultiActionController来处理。MultiActionController的默认处理方法为list，即罗列当前系统中与组相关的信息列表。这时，我们进入了groupAdmin.jsp渲染的视图画面，该画面有创建、更新和删除三个提交按钮，分别对应操作员组的创建、更新以及删除操作。任何一个提交按钮触发的表单提交，都将再次映射给我们的MultiActionController处理。我们的MultiActionController将根据提交的参数来决定调用哪个方法进行处理工作。总之，到目前为止，不管是最初的list请求还是之后整个表单的提交（存在三种提交分支情况），最终处理都是经由MultiActionController进行的。MultiActionController对逻辑相近的一组Web请求进行管理的特点在这里彰显无遗。

MultiActionController的不同处理方法执行之后，会根据当前场景将处理流程重定向到后继不同的分支流程上。比如，创建或者更新处理请求最终将被导向相应的表单页面进行后继操作，而这已经是后话了。所以，我们先将这部分流程的实现搁置一边，回头看看对应操作员组权限管理的MultiActionController是如何实现的吧！

- 操作员组权限管理实现流程概略

(1) 构建业务层支持。有了之前的基础，构建当前流程我们就惜些笔墨，所以，应该如何构建一个服务对象以便对操作员组进行管理等细节就不做详述了，只要我们达成了“需要一个IGroupAdminService和IGroupAdminService的实现类”这样的共识即可。如何将它们添加到相应的顶层

WebApplicationContext我想不必多提了，毕竟，我们此行的主要目的应该是MultiActionController。

(2) 构建MultiActionController。使用MultiActionController进行一组Web请求的处理有两种实现方式：

- 继承MultiActionController；
- 为MultiActionController提供一个委派对象。

为MultiActionController提供一个委派对象的好处在于，委派对象不需要继承任何父类或者接口。可以说是"干净利落，无牵无挂"，所以，我们就用它了。

我们初步决定使用ParameterMethodNameResolver对请求方法进行映射，具体来说，就是为methodParamNames指定create、update和delete作为要映射的参数，同时以list作为默认的方法，配置如下所示：

```
<bean id="methodNameResolver" ➥
class="org.springframework.Web.servlet.mvc.multiaction.ParameterMethodNameResolver">
    <property name="methodParamNames" value="create,update,delete"/>
    <property name="defaultMethodName" value="list"></property>
</bean>
```

这样我们所提供的委派对象就需要实现list、create、update和delete对应的Web请求处理方法。实现后的委派对象定义如代码清单24-8所示。

代码清单24-8　GroupAdminDelegate委派对象定义代码摘录

```
public class GroupAdminDelegate {

    private IGroupAdminService groupAdminService;
    private String listViewName;
    private String createViewName;
    private String updateViewName;
    private String deleteSuccessViewName;

    public ModelAndView list(HttpServletRequest request, HttpServletResponse response) ➥
    throws Exception
    {
        List<AdminGroup> groups = getGroupAdminService().getAvailableGroups();
        ModelAndView mav = new ModelAndView(getListViewName());
        mav.addObject("groups", groups);
        return mav;
    }

    public ModelAndView create(HttpServletRequest request, HttpServletResponse response) ➥
    throws Exception
    {
        return new ModelAndView(getCreateViewName());
    }

    public ModelAndView delete(HttpServletRequest request, HttpServletResponse response, ➥
    AdminGroup group) throws Exception
    {
        getGroupAdminService().deleteGroup(group);
        ModelAndView mav = new ModelAndView(getDeleteSuccessViewName());
        return mav;
    }
```

```
    public ModelAndView update(HttpServletRequest request, HttpServletResponse ➥
    response,AdminGroup group) throws Exception
    {
        return new ModelAndView(getUpdateViewName(),"groupName",group.getGroupName());
    }

    // getter和setter方法定义等
}
```

为了突出重点，整个GroupAdminDelegate的实现忽略了部分细节上的东西，比如异常的处理，以及相应情况下的视图跳转等。你在实现真实环境下的MultiActionController逻辑的时候，应该将这些逻辑添加进去，比如，直接使用MultiActionController可以定义异常处理方法的能力（custom exception handler method）。

有了MultiActionController使用的委派对象实现和MethodNameResolver，我们就能够在WebApplicationContext中，为操作员组权限管理相关的Web请求提供可用的MultiActionController了。如代码清单24-9所示。

代码清单24-9 MultiActionController相关类配置代码示例

```
<bean name="/groupAdmin.do" ➥
class="org.springframework.Web.servlet.mvc.multiaction.MultiActionController">
    <property name="delegate" ref="groupAdminDelegate"/>
    <property name="methodNameResolver" ref="methodNameResolver"/>
</bean>

<bean id="methodNameResolver" ➥
class="org.springframework.Web.servlet.mvc.multiaction.ParameterMethodNameResolver">
    <property name="methodParamNames" value="create,update,delete"/>
    <property name="defaultMethodName" value="list"></property>
</bean>

<bean id="groupAdminDelegate" class="cn.spring21.simplefx.controllers.GroupAdminDelegate">
    <property name="groupAdminService" ref="groupAdminService"/>
    <property name="listViewName" value="groupAdmin"/>
    <property name="createViewName" value="redirect:createGroup.do"/>
    <property name="updateViewName" value="redirect:updateGroup.do"/>
    <property name="deleteSuccessViewName" value="redirect:groupAdmin.do"/>
</bean>
```

通过MultiActionController的delegate属性，我们指定了它要使用的委派对象。Multi-ActionController内部将使用反射机制调用相应MethodNameResolver所返回的处理方法。

> **提示** 指定的逻辑视图名可以添加相应的前缀（prefix），比如redirect:viewName，将以重定向（redirect）的形式跳转到相应的视图。有关视图的转发（forward）和重定向（redirect），我们将在稍后ViewResolver和View部分详细介绍。

(3) 构建组权限管理默认视图。默认到/groupAdmin.do的Web请求将被引导到groupAdmin.jsp模板所渲染的视图画面。在这里，我们将根据情况选择要对操作员组做哪些操作，最终视图显示类似图24-8所示。

但不管具体操作是什么，最终该页面提交的请求将同样由我们的MultiActionController进行处理。

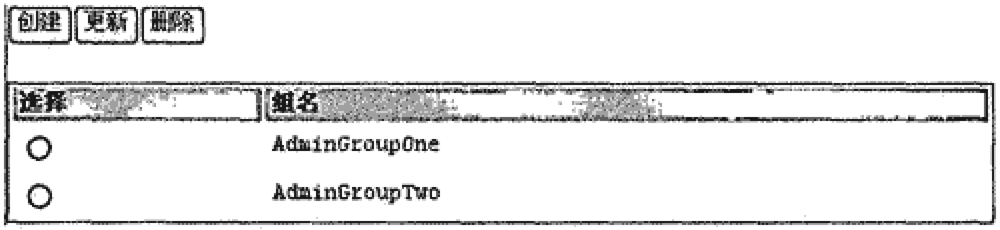

图24-8　groupAdmin页面显示

我们使用了DisplayTag进行了信息列表的输出。代码清单24-10中是groupAdmin.jsp模板文件的具体内容。

代码清单24-10　groupAdmin.jsp视图模板代码示例

```
<jsp:root version="1.2" xmlns:jsp="http:// java.sun.com/JSP/Page" ➥
xmlns:display="urn:jsptld:http:// displaytag.sf.net">
  <jsp:directive.page contentType="text/html; charset=UTF8" />
  <jsp:include page="/WEB-INF/jsp/header.jsp" flush="true" />

  <jsp:body>
      <form action="groupAdmin.do" method="post">
      <input name="create" type="submit" id="create" value="创建" />
      <input name="update" type="submit" id="update" value="更新" />
      <input name="delete" type="submit" id="delete" value="删除" />
      <display:table name="groups" id="group">
          <display:column title="选择"><input name="groupName" type="radio" ➥
          value="${group.groupName}" /></display:column>
          <display:column title="组名" property="groupName"></display:column>
      </display:table>
      </form>
  </jsp:body>
</jsp:root>
```

希望你能够在该模板的基础上进行完善，以进一步提升用户的体验，比如，如果用户要执行更新/删除操作，应该选取相应的组，否则，我们应该给予友好的提示。

MultiActionController可以帮助我们集中管理一组Web请求的处理逻辑，但因为需要通过反射调用相应的处理方法，所以，自身也就失去了编译期间检查等好处。另外，虽然MultiActionController也提供了数据绑定和数据验证功能，但并没有在表单的处理上做更多的流程定义，在允许开发人员有更多的发挥余地的同时，也可能引入风险：

- 滥用MultiActionController，进而导致同一个MultiActionController处理的职责过多体积臃肿而难以维护；
- 表单数据的处理上可能每个开发人员都有一套单独的处理方式，造成后期维护困难。

总之，要在合适的场景使用MultiActionController。如果涉及单一复杂的表单交互，尽量使用即将上场的SimpleFormController进行处理。

24.2.3　SimpleFormController

作为"规范操作派"当门大弟子，SimpleFormController首先继承了该派"掌门人"BaseCommandController的自动数据绑定和通过Validator的数据验证功能。不过，BaseCommandController也只传授SimpleFormController以上两种主要能力，却没有传授打通"经脉之法"。还好，BaseCommandController后继有人，AbstractFormController在BaseCommandCon-

troller的“武功”基础上，发展了一套模板化的form处理流程。至此，从数据的封装，验证，再到处理流程的模板化，整个规范化体系即告建立完成。而SimpleFormController以及稍后为各位详细介绍的AbstractWizardFormController从一出生，就被纳入了这套规范化体系。SimpleFormController专门面向单一表单的处理，而AbstractWizardFormController则提供多页面向导的交互能力。

要让SimpleFormController帮助我们简化Web请求处理工作，我们首先需要了解它。下面就让我们详细看一下SimpleFormController与生俱来的三种主要功能吧！

1. 了解数据绑定

在Web应用程序中使用数据绑定的最主要好处就是，我们再也不用自己通过request.getParameter(String)方法遍历获取每个请求参数，然后根据需要转型为自己需要的类型了。Spring MVC提供的数据绑定功能帮助我们自动提取HttpServletRequest中的相应参数，然后转型为需要的对象类型。我们唯一需要做的，就是为数据绑定提供一个目标对象，这个目标对象在Spring中称为Command对象（之前说过，它类似于Struts 1.1中FormBean的概念），此后的Web处理逻辑直接同数据绑定完成的Command对象打交道即可。

对于BaseCommandController及其子类来说，我们可以通过它们的commandClass属性设置数据绑定的目标Command对象类型，如下所示：

24

```
<bean id="commandController" class="..AnySubClassOfBaseCommandController">
    <property name="commandClass" value="..Command"/>
</bean>
```

或者直接在子类的构造方法中直接设定，如下所示：

```
public class BindingDemoController extends SimpleFormController {
    public BindingDemoController(){
        setCommandClass(Command.class);
        // 进行其他必要设置
    }
    ...
}
```

那么，Spring MVC在数据绑定过程中，是如何将请求参数绑定到我们所指定的Command对象上的呢？

有关数据绑定的过程，我们可以简单概括如下。

(1) 在Web请求到达之后，Spring MVC某个框架类将提取当前Web请求中的所有参数名称，然后遍历它，以获取对应每个参数的值，获取的参数名与参数值通常放入一个值对象（PropertyValue）中。最终我们将拥有所有需要绑定的参数和参数值的一个集合（Collection）。

(2) 有了即将绑定到目标Command对象的数据来源之后，我们即可将这些数据根据Command对象中各个域属性定义的类型进行数据转型，然后设置到Command对象上。

在这个过程中我们将碰到我们的老朋友BeanWrapperImpl，还记得IoC容器讲解内容中BeanWrapper第一次登场的情景吗？BeanWrapperImpl会将Command对象纳入自身管理范围，如下所示：

```
BeanWrapper beanWrapper = new BeanWrapperImpl(command);
```

然后比照参数名与Command对象的属性对应关系，以进行参数值到Command对象属性的设置，而参数值与Command对象属性间类型差异性的转换工作，则由BeanWrapperImpl所依赖的一系列

自定义PropertyEditor负责[1]。如果BeanWrapperImpl所使用的默认的PropertyEditor没有提供对某一种类型的支持，我们也可以添加自定义的PropertyEditor。这些内容你应该已经熟悉了，而我要澄清的是，在绑定过程中，参数名称与Command对象属性之间的对应关系是如何确定的。

假设我们有Command对象定义如代码清单24-11所示。

代码清单24-11　Command对象相关定义类代码示例

```
public class CustomerMetadata
{
    private String address;
    private String zipCode;
    private List<PhoneNumber> phoneNumbers = new ArrayList<PhoneNumber>();

    public CustomerMetadata()
    {
        phoneNumbers.add(new PhoneNumber());
    }

    // getter和setter方法定义
}

public class PhoneNumber {
    private String areaCode;
    private String number;

    public String getAreaCode() {
        return areaCode;
    }
    public void setAreaCode(String areaCode) {
        this.areaCode = areaCode;
    }
    public String getNumber() {
        return number;
    }
    public void setNumber(String number) {
        this.number = number;
    }
    // toString()等方法定义
}
```

那么，为了能够让请求参数对应到Command对象的相应属性，我们需要按照如下格式定义可能发送的参数名称：

```
<input type="text" name="address"/>
<input type="text" name="zipCode"/>
<input type="text" name="phoneNumbers[0].number"/>
```

也就是说，参数的名称对应Command对象的属性名称。对于嵌套的集合类型，Spring MVC还允许使用类似于JSTL和OGNL（Object-Graph Navigation Language）的语法来定义简单的表达式，例如phoneNumbers[0].number。表24-1给出了部分表达式的说明。

---

① 你可以在org.springframework.beans.PropertyEditorRegistrySupport的代码中发现所有默认注册的PropertyEditor类型。

表24-1 部分表达式实例

表 达 式	说 明
mapping[key].number mapping['key'].number mapping["key"].number	对于Map类型的集合，[]运算符（请允许我这么说）可以通过三种形式指定的key匹配Command对象中Map类型集合对应的元素
list[0].number array['1'].number list["2"].number	与Map类型对应的表达式中的[]运算符所接受的参数不同，对于List或者Array类型的集合，[]运算符现在接受的是索引值（index）而不是key

只要我们按照以上规则指定合适的参数名称，BaseCommandController及其子类就可以保证这些参数的参数值将被正确地绑定到Command对象及其嵌套对象上。不过，在使用嵌套表达式的时候，需要注意，必须保证中间环节对象不能为空。如果我们在构造CustomerMetadata的时候不为phoneNumber属性添加至少一个元素的话，那么phoneNumbers[0].number将抛给你“亲切的”NullPointerException。因为这个时候你的phoneNumber列表中根本就不存在第一个元素(phoneNumbers[0]==null)，何来null.setNumber(..)之理呢？

> 注意 有关绑定的参数表达式与Command对象属性之间的设置，可以进一步参照Spring参考文档的“Bean manipulation and the BeanWrapper”内容。

从参数获取到参数转型并绑定到Command对象，这整个流程最终以org.springframework.web.bind.ServletRequestDataBinder的形式进行了封装。该类常见使用代码如下所示：

```
CustomerMetadata customerMetadata = new CustomerMetadata();
ServletRequestDataBinder dataBinder = new ServletRequestDataBinder(customerMetadata);
// 如果必要的话，可以先进行dataBinder.registerCustomEditor(...)
dataBinder.bind(request);
Errors errors = binder.getErrors();
...
```

当然，我们不用自己去写这些几乎固定套路的代码，BaseCommandController及其子类将负责调用该类完成整个绑定过程，并返回已经绑定数据的Command对象。而之后，我们只需要使用数据绑定完成的Command对象进行后继处理就行了，比如，即刻对Command对象绑定后的数据进行数据验证工作。

2. Spring框架数据验证简介

Spring框架提供的数据验证支持并不只是局限于Spring MVC内部使用，从数据验证类所在的包名就能看出来，即org.springframework.validation。只要愿意，我们完全可以在独立运行的应用程序中使用Spring的数据验证功能。

Spring数据验证框架核心类为org.springframework.validation.Validator和org.springframework.validation.Errors，Validator负责实现具体的验证逻辑，而Errors负责承载验证过程中出现的错误信息，二者之间的纽带则是Validator接口定义的主要验证方法validate(target,errors)。从如下Validator的定义中可以看出这一点：

```
public interface Validator {
    boolean supports(Class clazz);
    void validate(Object target, Errors errors);
}
```

Validator具体实现类可以在执行验证逻辑的过程中，随时将验证中的错误信息添加到通过方法参数传入的Errors对象内。这样，验证逻辑执行完成之后，我们就可通过Errors检索验证结果啦！至于Validator接口中的support(Class)方法定义，是为了进一步限定Validator实现类的职责。除非你想压缩“人力成本”，把所有的数据验证工作全都交给一个Validator实现类去做。不过，那样的话，这个Validator就惨了！

正如我们所见到的那样，Validator接口定义很简单，而实现一个Validator实际上也同样简单。在CustomerMetadata完成数据绑定之后，我们需要对其绑定后的数据做进一步的验证，比如，提供的address和zipCode不能为空。如果提供了电话，那么，电话号码必须非空而且是数字等。这时，我们就需要为CustomerMetadata提供一个Validator实现类，代码清单24-12给出了对应的Validator实现类。

代码清单24-12 CustomerMetadata相关的Validator实现类代码示例

```
public class CustomerMetaDataValidator implements Validator {

    private PhoneNumberValidator phoneNumberValidator;

    public CustomerMetaDataValidator(PhoneNumberValidator phoneNumberValidator)
    {
        this.phoneNumberValidator = phoneNumberValidator;
    }

    public boolean supports(Class clazz) {
        return ClassUtils.isAssignable(clazz, CustomerMetadata.class);
    }

    public void validate(Object target, Errors errors) {
        CustomerMetadata customerMetadata = (CustomerMetadata)target;
        ValidationUtils.rejectIfEmpty(errors, "address", "address.empty");
        ValidationUtils.rejectIfEmpty(errors, "zipCode", "zipcode.empty");

        List<PhoneNumber> phoneNumbers = customerMetadata.getPhoneNumbers();
        if(CollectionUtils.isNotEmpty(phoneNumbers))
        {
            for(int i=0,size=phoneNumbers.size();i<size;i++)
            {
                PhoneNumber phone = (PhoneNumber)phoneNumbers.get(i);
                errors.pushNestedPath("phoneNumbers["+i+"]");
                ValidationUtils.invokeValidator(phoneNumberValidator, phone, errors);
                errors.popNestedPath();
            }
        }
    }
}


public class PhoneNumberValidator implements Validator {

    public boolean supports(Class clazz) {
        return ClassUtils.isAssignable(clazz, PhoneNumber.class);
    }

    /**
     * errors.reject(errorCode);  // Global Error with Type of ObjectError
     * errors.rejectValue(filed,errorCode,..); // Field Error with Type of FieldError
```

```
     */
    public void validate(Object target, Errors errors) {
        PhoneNumber phoneNumber = (PhoneNumber)target;
        if(phoneNumber == null)
           errors.reject("errorCodes.phoneNumber.is.null");

        if(!StringUtils.isNumeric(phoneNumber.getAreaCode()))
           errors.rejectValue("areaCode","areaCode.not.numeric", "areaCode cann't be empty!");

        if(!StringUtils.isNumeric(phoneNumber.getNumber()))
           errors.rejectValue("number", "phoneNumber.not.numeric");
    }
}
```

当然，也可以将CustomerMetadata所持有的PhoneNumber的验证逻辑一并合在CustomerMetaDataValidator中实现。不过，为了能够各司其职，还是为PhoneNumber的验证逻辑提供一个单独可复用的Validator实现比较合适。

初看两个Validator实现可能有些一头雾水，我们一个一个进行分析，如下所述。

PhoneNumberValidator中需要关注的地方。我们之所以先从PhoneNumberValidator开始，是因为这个Validator实现比较单纯，可以演示通常的Validator实现场景。一个Validator实现类，首先应该通过supports(Class)方法界定自身负责的验证范围，然后才是具体的验证逻辑。“细化社会分工”，这很重要！

具体的验证逻辑通常针对两种数据实体，一种就是被验证对象本身，另一种就是被验证对象的相应属性。如果被验证对象本身都不能通过验证，那么，这种错误我们称之为Global Error。这时，我们使用Errors的reject(String,..)这组方法，向Errors中添加相应的错误信息。如果被验证对象的某个属性域不能够通过验证，那么，我们称这种错误为Field Error。这时，我们要使用Errors的rejectValue(String,String,..)这组方法向Errors中添加相应的错误信息。

reject(String,..)方法第一个参数是错误信息对应的errorCode，而rejectValue(String,String,..)方法第一个参数是未能通过验证的属性域的名称。第二个参数才是对应错误信息的errorCode。这两个方法的差别就是这样，现在我们再回头看PhoneNumberValidator的实现，应该可以豁然开朗了。

解读CustomerMetaDataValidator。CustomerMetaDataValidator中有两个比较重要的点需要我们关注，如下所述。

- 对于不能通过数据验证逻辑的属性域，最基本的做法是通过Errors对象的rejectValue(String,String,..)方法将其信息添加到Errors对象。不过，如果对应某个对象域的验证仅限于“是否为空”这样的逻辑的话，我们也可以使用ValidationUtils这个工具类所提供的一组rejectIfEmpty(..)方法来达到同样的目的。这也就是我们在验证CustomerMetadata对象的address和zipCode两个对象域的时候使用ValidationUtils的原因。
- 如果要对当前对象的嵌套属性域进行验证，我们需要在调用对应嵌套对象的Validator实现类之前，调用Errors的pushNestedPath(String path)方法来明确当前被验证对象的上下文路径，并且在调用之后，通过popNestedPath()恢复之前的上下文路径。否则，当Errors对象绑定对应嵌套对象属性的错误信息的时候，会认为该属性是上层目标对象上的属性，这时就会出现绑定上的异常了。如果我们不使用pushNestedPath(String path)方法，Errors在记录number对应的错误信息的时候，同时需要记录对应该属性的值，那么它就会根据当前属性域对应的表达式到Command对象上获取。可是当它根据number到Customer-

Metadata上查找的时候，就会发现，根本就找不到CustomerMetadata上有一个叫做number的属性域，自然就会抛出异常。可是，如果在此之前，我们通过pushNestedPath(String path)方法改变Errors注册属性域错误信息所使用的上下文路径，比如，变成phoneNumber[0]，那么，当Errors注册number对应的错误信息的时候，就会以phoneNumber[0].number到CustomerMetadata获取对应的属性值。现在，自然就没有问题了。

基本上，实现一个Validator要注意的地方也就这些了。我想，了解了以上这些实现细节之后，让你再去实现任意一个Validator，你应该不会“手软”了吧！

代码清单24-13给出的是调用具体Validator实现类执行数据验证逻辑的普通场景。

代码清单24-13　Validator常见的使用代码示例

```
CustomerMetaDataValidator validator = new CustomerMetaDataValidator(new ➥
PhoneNumberValidator());
CustomerMetadata md = new CustomerMetadata();
// 为Command对象设置测试数据

BindException errors = new BindException(md,"customerMd");
ValidationUtils.invokeValidator(validator, md, errors);
assertTrue(errors.hasErrors());

Map map = errors.getBindingResult().getModel();
BindingResult result = ➥
(BindingResult)map.get("org.springframework.validation.BindingResult.customerMd");
// 遍历BindingResult以取得可用的错误信息
```

我们只需要构造一个具体的Command对象实例（CustomerMetadata）以及一个Errors实例（BindException），然后通过ValidationUtils调用对应的Validator实现类（不使用ValidationUtils直接调用Validator的validate()方法也一样）。调用完成之后，即数据验证完成，如果存在验证错误（可以通过errors.hasErrors()获知），我们可以遍历之前传入的errors以获取相应的错误信息，然后根据具体应用程序的场景做后继处理。

在Spring MVC中，以上Validator实现类的执行以及后继错误信息的处理，将由BaseCommandController或者其子类接管。我们通常不需要操心这些细节，唯一需要我们做的是，通过相应的setter方法，为BaseCommandController或者其子类（比如当前的SimpleFormController）提供需要使用的Validator实现，如下所示：

```
<bean id="commandController" class="SubClassOfBaseCommandController">
    <property name="validators">
        <list>
            <ref bean="validator1"/>
            <ref bean="validator2"/>
            ...
        </list>
    </property>
</bean>
```

在实际的项目开发过程中，除了使用以上编程方式实现数据验证工作，我们还可以借助于Commons Validator（http://commons.apache.org/validator/）或者Valang（http://opensource.atlassian. com/confluence/spring/display/MODULES/Using+Valang+validator）实现声明式的数据验证，详细内容可以参考*Expert*

*Spring MVC and Web Flow*一书，其中对使用Valang进行声明式数据验证进行了详尽的介绍，我们就不在这里进行赘述了。

3. 深入表单（form）处理流程

SimpleFormController及其父类AbstractFormController最主要的一个特点就是对表单的处理流程进行了统一。可以毫不夸张地说，只要掌握了AbstractFormController以及其子类的表单处理流程，我们就基本掌握了整个本派（当然指的是“规范操作派”）武功之精髓。

AbstractFormController以模板方法模式从顶层界定了主体上的流程处理逻辑，而处理流程中某些特定的动作则留给其子类SimpleFormController（以及AbstractWizardFormController）实现。但以模板方法模式实现的整个流程控制，并非真得就像“模板”所听起来的那样死板，我们可以通过覆写其中的某些方法以添加自定义的行为逻辑，体现了整个流程的可扩展性和灵活性。

对于整个模板流程逻辑，AbstractFormController和SimpleFormController的Javadoc中有详细的描述。但为了能够帮助大家更容易地了解这个流程，以便之后的开发过程中能够灵活驾驭SimpleFormController，我们将结合图24-9详细讲述该“武术套路”。

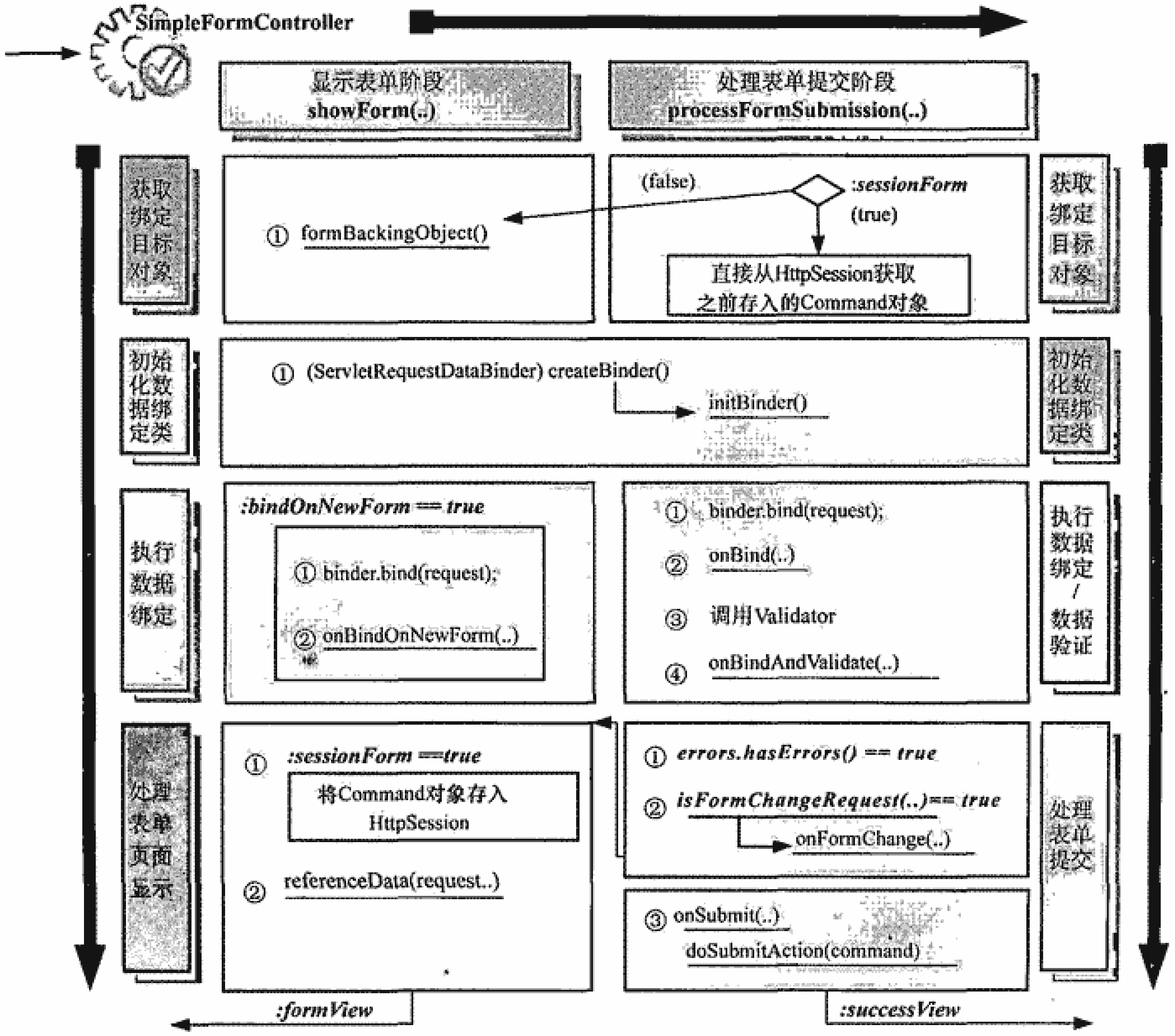

图24-9　SimpleFormController处理流程图

我们将AbstractFormController的流程逻辑整合到SimpleFormController的最终流程逻辑中，从而可以看到一个完整的流程实现逻辑（AbstractFormController只是提供了表单处理流程的主体框架）。下面是我们对整个表单处理流程的描述。

在Web请求到达SimpleFormController之后，SimpleFormController将首先通过方法isFormSubmission(request)判明当前请求是否为表单提交请求，isFormSubmission (request)的默认实现如下所示：

```
protected boolean isFormSubmission(HttpServletRequest request) {
    return "POST".equals(request.getMethod());
}
```

也就是说，只要是以POST形式发送的Web请求，SimpleFormController将认为当前Web请求即为表单提交。我们可以通过覆写该方法以改变默认的判定标准。

整个表单处理流程将以isFormSubmission(request)的判定结果为基准，划分为“表单显示阶段”和“处理表单提交阶段”两个逻辑处理区段。如果isFormSubmission(request)返回false，通常表示初次Web请求，这时我们需要为用户显示相应的交互表单，这也就是“表单显示阶段”将要做的事情。否则，将认为用户已经在表单中编辑完数据，需要处理，SimpleFormController将启用“处理表单提交阶段”流程处理逻辑。下面是我们对这两个阶段流程处理逻辑的详细分析。

- *表单显示阶段流程分析*

(1) 创建或者获取表单对应的Backing Object。我也不知道为什么将该对象叫做form BackingObject。实际上它就是对应绑定表单数据的Command对象。不过，我想我们只要知道这个对象干什么的就可以了，没有必要在名字上深究。

此时，formBackingObject()方法将默认通过反射实例化我们为SimpleFormController指定的Command对象实例。对于像“新添加某个实体到系统数据库”这样的场景来说，formBackingObject()方法的默认行为就足够了，对于之前不存在的实体，添加到系统之前，它本来就是一个“空白”对象嘛！

但是某些时候formBackingObject()的默认行为不能符合当前场景需求。比如，如果我要更新一个实体的信息，那么，在为用户显示表单的时候，就应该将要更新的实体的数据加载到表单中，以便用户在原有数据的基础上进行更改。这时，我们就需要覆写formBackingObject()方法，改为自己管理Form Backing Object的初始化。比如，从数据库中加载数据到将要返回给表单显示的Form Backing Object中。

> **注意** *图24-9中所有像formBackingObject()方法那样下面加横线的方法，都是可以覆写以添加自定义行为的地方，这是SimpleFormController为我们提供的扩展点。*

(2) 初始化DataBinder。因为BaseCommandController门下所有门徒都采用将请求参数绑定到Command对象的处理方式，这当然也包括SimpleFormController，所以，我们就得在执行请求参数到Command对象的数据绑定之前，初始化一个可用的DataBinder实例。具体点儿说，就是我们在24.2.3节的第1小节所介绍的ServletRequestDataBinder。

在有了一个可用的ServletRequestDataBinder之后，我们可以对其进行定制，比如添加自定义的PropertyEditor以支持某些特殊数据类型的数据绑定，或者排除某些不想绑定的请求参数，这些定制行为可以通过覆写initBinder()方法引入，例如：

```
@Override
protected void initBinder(HttpServletRequest request, ➥
ServletRequestDataBinder binder) throws Exception {
    PropertyEditor propertyEditor = ...;
    binder.registerCustomEditor(SpecialClass.class, propertyEditor);
    binder.setDisallowedFields(new String[]{"parameter1","parameter2"});
}
```

ServletRequestDataBinder准备完毕之后，就可以随时用于数据绑定操作了。

(3) 执行数据绑定。这一步只有在bindOnNewForm属性被设置为true的情况下才会触发执行。bindOnNewForm默认值是false，所以，通常情况下是不会在显示表单之前执行任何绑定操作的。但是，如果针对SimpleFormController发起的初次请求中存在某些参数，并且我们想将它们绑定到创建好的Command对象，那么就可以在SimpleFormController的构造方法或者bean定义上设置bindOnNewForm为true（如下代码清单24-14所示），剩下的事情就交给SimpleFormController去管了。

代码清单24-14 bindOnNewForm属性设置代码示例

```
public class YourFormController extends SimpleFormController
{
    public YourFormController()
    {
        setBindOnNewForm(true);
        // ……其他设置，如setCommandClass(..)
    }
}
或者
<bean id=".." class="..YourFormController">
        <property name="bindOnNewForm" value="true"/>
</bean>
```

在将bindOnNewForm设置为true触发了数据绑定操作之后，ServletRequestDataBinder的bind(request)方法将被调用，从而完成要求的数据绑定。之后，如果我们想对绑定后的Command对象做进一步的定制，可以覆写onBindOnNewForm(..)方法以插入自定义逻辑。

(4) 处理表单的显示。在最终显示表单页面之前，还有最后一点儿工作要做，如下所述。

- 如果SimpleFormController的sessionForm属性被设置为true的话（实际上是其父类AbstractFormController定义的该属性），我们会将绑定后的Command对象存入HttpSession。这样在提交表单之后，也就是“处理表单提交阶段”，可以重新获取该Command对象，而不是又重新生成一个（那样的话，之前对Command对象的数据定制将不复存在）。
- Form Backing Object更多的面向表单的各个字段，与表单中的字段形成双向的绑定关系。比如AdminGroup的groupName属性对应表单中的<input name="groupName" type="radio" value="${group.groupName}"/>。但某些情况下，表单页面的显示还需要Form Backing Object所包含的数据之外的信息，比如，如果我们的表单中需要提供某种动态信息的可选择下拉列表，那么，这时可能就需要从数据库中抽取将构建下拉列表所需要的数据。这种类型的数据信息可以通过覆写referenceData(..)方法来提供，SimpleFormController将会把通过referenceData(..)返回的模型数据添加到即将返回的ModelAndView中。

如果你还是无法搞清楚Form Backing Object中的数据与通过referenceData(..)返回的数据有何区别，我们来看一个实例。

假设我们的下拉列表框显示一年中的12个月，在referenceData(..)中，我们可能需要添加如代码清单24-15所示代码逻辑。

代码清单24-15 referenceData()方法应用代码示例

```
@Override
protected Map referenceData(HttpServletRequest request, Object command,
        Errors errors) throws Exception {
    Map data = new HashMap();

    List<String> months = new ArrayList<String>();
    months.add("January");
    months.add("February");
    // 添加其他月份……
    data.put("months", months);
    return data;
}
```

视图模板中将取得referenceData(..)所返回的数据进行显示，如代码清单24-16所示。

代码清单24-16 使用referenceData()方法返回数据的视图模板代码示例

```
<select name="monthOfCreation">
    <c:forEach var="month" items="${months}">
         <c:choose>
                    <c:when test="${month eq monthOfCreation}">
                        <option value="${month}" selected="true">${month}</option>
                    </c:when>
                    <c:otherwise>
                        <option value="${month}">${month}</option>
                    </c:otherwise>
                </c:choose>
    </c:forEach>
</select>
```

referenceData(..)所返回的应该是对应要遍历的数据，而我们的Form Backing Object中存放的实际上只是遍历的数据中的某个值。具体到我们的示例就是，视图模板中${monthOfCreation}对应的值，应该来自我们的Form Backing Object，而${months}则来自referenceData(..)方法[①]。

总地来说，referenceData(..)是返回可供选择的数据列表。而Form Backing Object通常只是保存数据列表中选择后的那个值。当然，将这样的数据一并添加到Form Backing Object返回也是可以的，但各自的语义也就有些含混了。

在以上准备工作完成之后，SimpleFormController就会将所有这些“搜刮”来的数据一并放到某个ModelAndView实例中并返回（该ModelAndView将使用formView属性所指定的逻辑视图名）。剩下会发生什么我想你已经知道了。

- 处理表单提交阶段的流程分析

(1) 获取要绑定的目标对象。用户提交表单之后，我们首先会判断SimpleFormController的sessionForm属性是否为true。如果是，我们将从HttpSession中获取之前已经存入的Command对象。否则，与“表单显示阶段”一样，通过formBackingObject()方法重新生成一个Command对象以备后继数据绑定之用。

实际上，除了做一下简单的条件判断，该步骤与“表单显示阶段”的同一步骤目的是相同的，都

① 当然，以上模板代码只是为了演示，实际开发中，直接使用Spring提供的检签库（tag lib）要方便得多。

是为了能够为后继操作提供一个Command对象。

(2) 初始化DataBinder。这一步与“表单显示阶段”同一步骤完全相同，没有必要多说了吧？

(3) 执行数据绑定/数据验证。在用户通过表单提交数据之后，我们需要将这些新的数据绑定到Command对象。在绑定的实现上，与“表单显示阶段”同一步骤原理相同，只不过，绑定完成后为我们提供的扩展点是onBind(..)方法。我们可以覆写该方法对绑定后的Command对象做进一步的定制。

与显示表单阶段不同，完成数据绑定之后，还需要对用户提交的数据进行验证，否则，非法的数据将有可能危害整个系统的安全。通常，数据验证可以在两个地方进行，一个是客户端，通过客户端脚本（比如Javascript）做初步的数据验证，各浏览器对同一脚本的支持存在差异先不提，最主要的问题是，如果有意，用户可以轻易避开客户端通过脚本提供的这段数据验证“安全网”，所以，我们得在另一个地方拉起最主要的一道数据验证“安全网”，那就是在服务器端，而我们当前所处的位置，就是SimpleFormController为我们提供的构筑服务器端数据验证“安全网”的地方。

数据验证逻辑的执行实际上是不需要我们关心的。要实现服务器端数据验证，我们所要做的只是为SimpleFormController指定一组用于对当前表单提交的数据进行数据验证的Validator实现类就行了，也就是说，我们是通过配置相应的Validator来提供具体的数据验证逻辑的。

虽然Validator的调用我们不能干预，但是，如果需要在数据验证完成后对Command对象做一些“手脚”，我们还是有机会的。通过覆写onBindAndValidate(..)方法，我们可以在继续剩下的逻辑之前插入一些自定义逻辑。

(4) 处理表单提交。胜利就在眼前，不过少安毋躁，在允许我们调用相应的业务对象处理最终的数据之前，SimpleFormController还要做进一步的“安检”工作。

SimpleFormController会首先检查数据验证后是否存在验证错误。如果存在验证错误，它将会把处理流程导向“显示表单阶段”的最后一步，也就说，重新显示表单页面。只不过，这时，表单页面除了显示最初的数据之外，还得显示相应的验证错误信息。

如果没有验证错误，SimpleFormController再检查一下当前Web请求是否只是一个FormChange-Request，也就是说，当前Web请求只是对表单的数据进行了一些变动，而不是最终的提交。这时，我们依然不能处理数据，而是需要将视图重新导向表单的显示页面，就与刚才出现验证错误时候我们所做的一样。不过，在将流程导向表单页面显示之前，我们可以通过覆写onFormChange()方法添加某些自定义逻辑，比如更改Command对象数据状态。

> **注意** SimpleFormController默认的isFormChangeRequest(..)返回false，即不支持该类型Web请求，我们可以通过覆写该方法以提供自定义的判定标准。

在以上这些检查通过之后，我们就可以真正地处理表单提交的数据了。SimpleFormController允许我们在如下两个地方提供表单数据的处理逻辑实现。

- 我们可以通过覆写方法doSubmitAction(Object command)来添加针对当前表单数据的处理逻辑，最常见的就是直接调用相应的服务对象。doSubmitAction(command)方法执行之后，SimpleFormController将构建一个ModelAndView将视图导向successView属性所指定的视图。
- 实际上，doSubmitAction(command)只是onSubmit(..)方法所公开的一个回调方法。当前者执行完毕之后，是由onSubmit(..)来构建相应的ModelAndView以“打扫战场”的。所以，如果感觉doSubmitAction(command)方法不能为我们提供更大的发挥余地，那么，

我们完全可以直接覆写onSubmit(..)方法来实现具体的处理逻辑。只不过，在最后要自己构建一个ModelAndView并返回。当然，通常情况下，直接覆写doSubmitAction(command)方法应该就足够了。

注意　SimpleFormController提供了重载的onSubmit(..)方法，我们可以根据需要选择覆写哪个。

在视图被导向successView属性所指定的页面之后，我们的表单处理流程就算结束了。

整个流程处理逻辑看起来可能很复杂，让人有些望而却步。但实际上，要实现一个SimpleFormController以处理与表单的交互很简单。通常也就是设置一下要绑定的Command对象，然后覆写一下doSubmitAction(command)方法即可。至于其他，像是否要添加Validator，是否要将Command对象存入session，以及要不要在表单显示之前就进行数据绑定之类的逻辑，完全可以根据应用程序的需要进行添加。代码清单24-17给出了一个最基本的SimpleFormController实现类。

代码清单24-17　简单的SimpleFormController实现类代码示例

```
public class HelloWorldFormController extends SimpleFormController {
    public HelloWorldFormController(){
        setCommandClass(YourCommandObject.class);
        setCommandName("optional");
    }

    @Override
    protected void doSubmitAction(Object command) throws Exception {
        System.out.println("Hello, we have got a command to process."+command.toString());
    }
}
```

当然，HelloWorld永远只是一个玩具，还是让我带你进入更真实一些的世界吧！

4. SimpleFormController应用演示

书接上回，话说我们为了演示MultiActionController的使用，而引入了操作员组管理的处理场景，当时虽然给出了整个的流程，但仅对使用MultiActionController实现的部分进行了介绍。现在我们要完成图24-7中整个流程图的后半部分实现。

在用户通过组管理画面groupAdmin.jsp提交“添加新操作员组”或者“更新某个操作员组”的请求之后，MultiActionController实现会将这两种请求转发给后继的Controller进行处理。正如我们所看到的那样，不管是要添加数据还是要更新数据，都需要首先为用户提供一个表单页面，然后再对用户通过表单提交的数据进行处理。这里所谓的后继Controller，通过继承SimpleFormController来实现是最为合适的方式。

下面是对“操作员组添加以及更新操作实现流程”的简短介绍。通过对比两个不同的SimpleFormController，以期望大家可以从中窥得SimpleFormController的各种奥秘以及使用方式。

- **添加新的操作员组实现流程**

我们先来看一下添加操作员组的视图画面，以便先对这个要实现的功能有一个感性的认识，如图24-10所示。

虽然组名是区分各个组的主要标志，但本质上来说，各个组之间的本质差别则是每个组所持有的权限。我们添加新组也好，更新旧组也好，说白了，是对每个组所持有的权限进行操作。

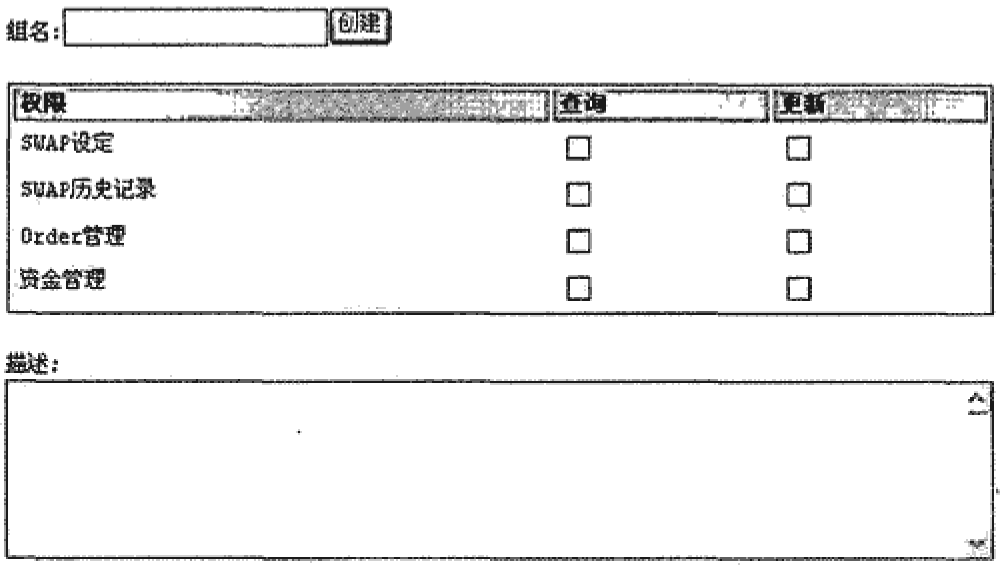

图24-10 添加新操作员组的视图页面

(1) 业务层逻辑说明。新建组的时候会有一套默认的权限序列供我们选择，这些默认的权限序列需要通过某个服务对象取得。所以，除了之前的IGroupAdminService及其实现类之外，现在还得提供一个IAuthorityService以及它的实现类，我们暂且提供如代码清单24-18所示的参考代码。

代码清单24-18 IAuthorityService及其实现类定义代码

```
public interface IAuthorityService
{
    List<AuthorityRule> getAvailableAuthorityRules();
}

public class MockAuthorityService implements IAuthorityService
{
    public List<AuthorityRule> getAvailableAuthorityRules() {
        List<AuthorityRule> rules = new ArrayList<AuthorityRule>();

        AuthorityRule swapSettingRule = new AuthorityRule();
        swapSettingRule.setId(1L);
        swapSettingRule.setAuthorityName("SWAP设定");
        rules.add(swapSettingRule);

        AuthorityRule swapHistoryMgtRule = new AuthorityRule();
        swapHistoryMgtRule.setAuthorityName("SWAP历史记录");
        swapHistoryMgtRule.setId(2L);
        rules.add(swapHistoryMgtRule);

        AuthorityRule orderMgtRule = new AuthorityRule();
        orderMgtRule.setAuthorityName("Order管理");
        orderMgtRule.setId(3L);
        rules.add(orderMgtRule);

        AuthorityRule cashflowMgtRule = new AuthorityRule();
        cashflowMgtRule.setAuthorityName("资金管理");
        cashflowMgtRule.setId(4L);
        rules.add(cashflowMgtRule);

        return rules;
    }
}
```

实际系统中，我们可能需要从数据库提取相应数据，以具体的实现类来替换掉当前提供的模拟数据实现类MockAuthorityService。

(2) 构建处理请求的SimpleFormController实现类。我们扩展SimpleFormController实现了一个GroupCreateController，用于显示新添加操作员组表单，并最终处理表单提交后的数据。GroupCreateController的定义见代码清单24-19。

代码清单24-19　GroupCreateController定义代码

```
public class GroupCreateController extends SimpleFormController
{
    private IGroupAdminService groupAdminService;
    private IAuthorityService authorityService;

    public GroupCreateController()
    {
        setCommandClass(AdminGroup.class);
        setCommandName("adminGroup");
    }

    @Override
    protected Object formBackingObject(HttpServletRequest request) ➥
    throws Exception {
        AdminGroup group = new AdminGroup();
        List<AuthorityRule> rules = getAuthorityService().getAvailableAuthorityRules();
        group.setRules(rules);
        return group;
    }

    @Override
    protected void doSubmitAction(Object command) throws Exception {
        getGroupAdminService().createGroup((AdminGroup)command);
    }
    // getter和setter方法定义
}
```

该实现可能需要关注的点有如下几个。

- 在构造方法内，我们通过setCommandClass(..)方法为GroupCreateController指定了将要使用的Command对象的类型，并且通过setCommandName(..)方法重新设置了GroupCreateController对应使用的Command对象的名称，视图模板将会根据该名称绑定Command对象数据到表单。setCommandName(..)的调用并非必须的，如果不明确设置Command名称的话，getCommandName()的值为“command”，也就是说，默认情况下，我们可以在视图中使用${command.property}的形式访问Command对象的属性值。
- 我们覆写了formBackingObject(..)方法，默认的formBackingObject(..)只是通过反射初始化了一个“空白”的Command对象。在我们的场景中，这样的行为是不够的。即使是新建组，我们也需要提供一组默认的权限供新建组时候选择。所以，我们通过覆写formBackingObject()方法来自定义Command对象的初始化，在实例化Command对象的基础上，同时为其添加了附加数据，即默认的权限序列。
- 在组建立成功之后，我们就会重新将视图导向最初的操作员组信息列表视图（groupAdmin.jsp），并不需要传递多余的数据。所以，我们只需要覆写doSubmitAction(command)方法来处理表单提交的数据即可。

GroupCreateController实现完成之后，我们需要将它添加到WebApplicationContext中，以便提供相应依赖的注入以及进一步细化对象配置，如下所示：

```
<bean name="/createGroup.do" ➥
class="cn.spring21.simplefx.controllers.GroupCreateController">
    <property name="groupAdminService" ref="groupAdminService"/>
    <property name="authorityService" ref="authorityService"/>
    <property name="validator" ref="adminGroupValidator"/>
    <property name="formView" value="createGroup"/>
    <property name="successView" value="redirect:groupAdmin.do"/>
</bean>
```

groupAdminService和authorityService两个属性分别对应当前Controller实现所依赖的服务对象，它们随着应用和流程不同而变化。而validator（或者validators）、formView和successView属性则是每个SimpleFormController实现类都应该提供的，尤其是formView和successView。SimpleFormController最终将根据这两个属性把视图导向表单页面以及表单处理成功后的页面。

表单提交后，通常需要提供相应的Validator进行数据的验证。如何实现Validator进行数据验证我们之前已经谈过了，这里只罗列一下我们的实现吧！如下所示：

```
public class AdminGroupValidator implements Validator {
    public boolean supports(Class clazz) {
       return ClassUtils.isAssignable(clazz, AdminGroup.class);
    }
    public void validate(Object target, Errors errors) {
       ValidationUtils.rejectIfEmptyOrWhitespace(errors, "groupName", "group.name.empty");
    }
}
```

我们的验证逻辑很简单，只是不允许新建组的组名为空而已。为了能够获取数据验证失败后的确切的错误信息，我们得在应用的顶层WebApplicationContext中定义一个MessageSource实现，名称为“messageSource”。这样，当需要显示错误信息的时候，在根据验证结果显示错误信息的代码中，就可以根据Errors中的错误码（errorCode）从WebApplicationContext中获取具体的错误信息了，如下所示：

```
<bean id="messageSource" class="org.springframework.context.support. ➥
ResourceBundleMessageSource">
    <property name="basename" value="messages"></property>
</bean>
```

关于MessageSource的更多信息可以回顾第二部分的相关内容。

(3) 构建视图。有了页面显示的初步印象，代码清单24-20给出的JSP模板代码看起来也应该很好理解了。

代码清单24-20 对应的JSP视图模板代码示例

```
<jsp:root version="1.2" xmlns:jsp="http:// java.sun.com/JSP/Page" ➥
xmlns:display="urn:jsptld:http:// displaytag.sf.net" ➥
xmlns:c="urn:jsptld:http:// java.sun.com/jsp/jstl/core" ➥
xmlns:form="urn:jsptld:http:// www.springframework.org/tags/form">
  <jsp:directive.page contentType="text/html; charset=UTF8" />
  <jsp:include page="/WEB-INF/jsp/header.jsp" flush="true" />

  <jsp:body>
```

```
        <form:form commandName="adminGroup">
            <form:errors path="*"></form:errors><br/>
            组名:<form:input path="groupName"/>
            <input type="submit" value="创建" name="create"/><br/>

            <display:table name="adminGroup.rules" id="row">
                <display:column property="authorityName" title="权限"></display:column>
                <display:column title="查询">
                    <form:checkbox path="rules[${row_rowNum-1}].canRead"/>
                </display:column>
                <display:column title="更新">
                    <form:checkbox path="rules[${row_rowNum-1}].canWrite"/>
                </display:column>
            </display:table>
            描述:<br/>
            <form:textarea path="description" cols="80"/>
        </form:form>
    </jsp:body>
</jsp:root>
```

我们混杂了displayTag和Spring的form标签库（Tag Library）两种标签库进行模板内容的表达。有关它们的使用和更多细节可以参考相应文档，这里就不再赘述了，需要强调如下几点。

- 因为我们在GroupCreateController的构造方法中，将Command对象的默认名称从“command”变成了“adminGroup”，所以，使用Spring的form标签的时候，需要通过commandName明确指定当前的Command对象的名称。
- 不管使用什么标签或者表达式语言（EL，Expression Language）来表达模板内容，如果想让Spring MVC帮我们将表单参数自动绑定到Command对象，一定要保证表单中的各个元素的name属性使用了正确的表达式，例如rules[0].canRead或者rules[0].canWrite。

- **更新现有操作员组信息的实现流程**

更新页面与新建组的页面几乎完全相同，唯一的区别只是提交按钮的名字从创建变成了更新，如图24-11所示。

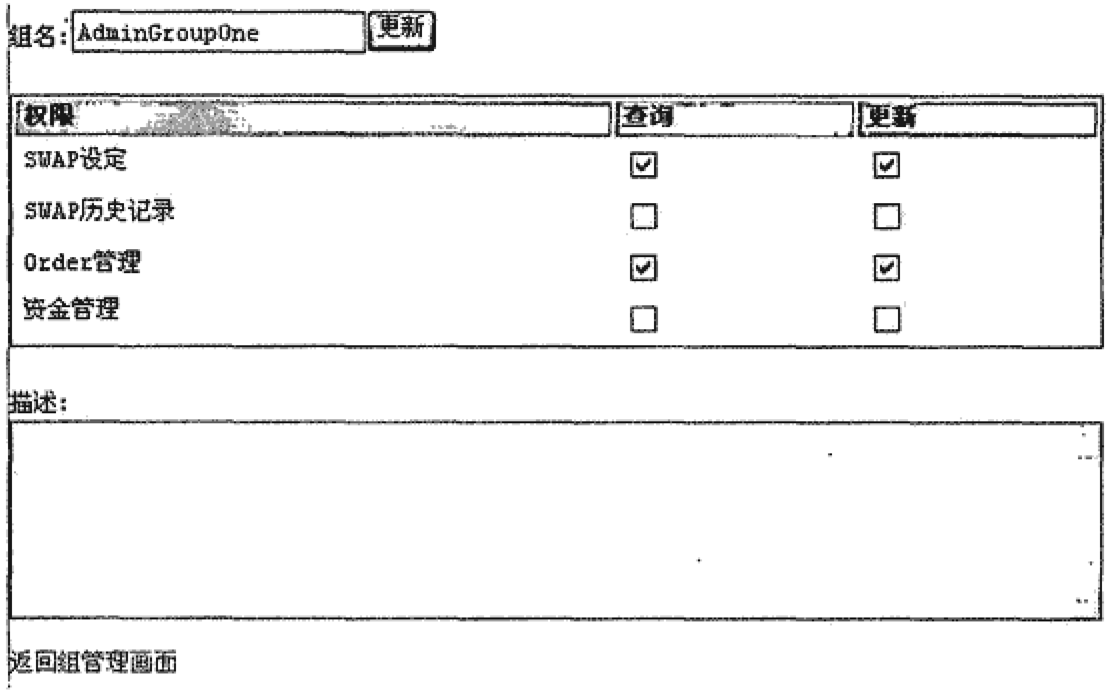

图24-11　更新管理组页面示意图

更新操作员组的SimpleFormController实现类使用的服务对象与GroupCreateController所使用的相同。所以，我们略过业务对象相关内容，直接开始构建需要的SimpleFormController实现类。

(1)构建处理表单的SimpleFormController。

当用户要更新一个组的信息的时候，他将在现有的“组信息列表”页面选择要更新的组，然后提交更新请求。接收到请求的MultiActionController将通过重定向的方式，将后继流程控制权转交给我们的SimpleFormController实现类。我们的实现类同样需要知道用户要更新的是哪一个组。所以，在MultiActionController的实现类GroupAdminDelegate中，我们将groupName添加到了返回的ModelAndView中。这样，重定向之后，发起请求的路径将类似于如下所示的样子：

```
http:// host:port/simplefx/updateGroup.do?groupName=AdminGroupOne
```

该路径对应的Web请求将被映射给我们的SimpleFormController处理。

与新添加一个组所使用的SimpleFormController实现类不同，更新用的SimpleFormController存在些许差异，其定义见代码清单24-21。

代码清单24-21 GroupUpdateController定义代码

```
public class GroupUpdateController extends SimpleFormController {

    private IGroupAdminService groupAdminService;
    private IAuthorityService authorityService;

    public GroupUpdateController()
    {
        setCommandClass(AdminGroup.class);
        setCommandName("adminGroup");
        setSessionForm(true);
        setBindOnNewForm(true);
    }

    @Override
    protected void onBindOnNewForm(HttpServletRequest request, Object command)
            throws Exception {
        AdminGroup group = (AdminGroup)command;
        List<AuthorityRule> rules = getAuthorityRulesFor(group);
        group.setRules(rules);
    }

    @Override
    protected ModelAndView onSubmit(Object command) throws Exception {
        AdminGroup group = (AdminGroup)command;
        getGroupAdminService().updateGroup(group);

        ModelAndView mav = new ModelAndView(getSuccessView());
        mav.addObject("groupName",group.getGroupName());
        return mav;

    }
    // getter方法、setter方法以及其他helper方法定义
}
```

使用的服务对象是一样的，不一样的地方是如下几点。

- 在构造方法中，我们启用了两个新的属性，即sessionForm和bindOnNewForm属性，如下所述。
  - 通过setSessionForm(true)，我们告知GroupUpdateController在处理“表单显示”到“表单提交”的整个过程中，将使用保存到HttpSession的同一个Command对象，而不是每次都新建一个。
  - 最初发起请求的路径为http:// host:port/simplefx/updateGroup.do?groupName=Admin GroupOne，这意味着我们需要获取groupName参数以确定对哪一个组进行数据更新。在过去，我们当然是通过request.getParameter("groupName")来获取参数值。但SimpleFormController的第一个特色就是支持参数到Command对象的绑定，所以，我们通过将bindOnNewForm设置为true来告知GroupUpdateController在“显示表单阶段”就进行参数绑定。
- 我们覆写了onBindOnNewForm(..)方法，该操作与设置bindOnNewForm属性为true是如影随形的。在参数绑定到Command对象之后，我们就得根据绑定到Command对象的参数值，来加载操作员组的原有信息到表单显示。现在看来，加载原有数据的逻辑也只能在onBindOnNew-Form()方法内添加了。不过，实际上，要达成类似的效果，我们也可以在formBacking-Object(..)中进行，如代码清单24-22所示。

代码清单24-22　通过formBackingObject()方法定制返回数据代码示例

```
@Override
protected Object formBackingObject(HttpServletRequest request) ➥
throws Exception {
    AdminGroup group = new AdminGroup();
    String groupName =
    ServletRequestUtils.getStringParameter(request, "groupName");
    group.setGroupName(groupName);
    List<AuthorityRule> rules = getAuthorityRulesFor(group);
    group.setRules(rules);
    return group;
}
```

  不过，在formBackingObject()中添加加载数据的逻辑，就要自己去处理参数获取等工作，有自动绑定参数的功能放着不用，多少有些违反门规的意思吧！

- 处理最终表单提交请求的时候，我们覆写了onSubmit(..)方法以取得更大的权限。因为像GroupCreateController那样只覆写一个doSubmitAction(..)方法在这里是不够的。我们希望在更新成功之后，依然将视图导向更新页面。这时，我们就得提供请求参数，以便GroupUpdateController能够根据这个参数正确显示页面数据。所以，我们得向返回的ModelAndView中添加参数，doSubmitAction(..)是不公开ModelAndView给我们的。

至于将GroupUpdateController添加到WebApplicationContext，就有些老生常谈了，如下所示：

```
<bean name="/updateGroup.do" class="cn.spring21.simplefx.controllers.GroupUpdateController">
    <property name="groupAdminService" ref="groupAdminService"/>
    <property name="authorityService" ref="authorityService"/>
    <property name="validator" ref="adminGroupValidator"/>
    <property name="formView" value="updateGroup"/>
    <property name="successView" value="redirect:updateGroup.do"/>
</bean>
```

(2) 构建视图。

组信息更新的视图与添加组的视图模板代码没有多大变化。如果可能，我们还可以进一步模块化两个视图的模板，组信息更新视图对应的定义代码如代码清单24-23所示。

代码清单24-23 组信息更新页面代码示例

```
<jsp:root version="1.2" xmlns:jsp="http:// java.sun.com/JSP/Page"
    xmlns:display="urn:jsptld:http:// displaytag.sf.net"
    xmlns:c="urn:jsptld:http:// java.sun.com/jsp/jstl/core"
    xmlns:form="urn:jsptld:http:// www.springframework.org/tags/form">
  <jsp:directive.page contentType="text/html; charset=UTF8" />
  <jsp:include page="/WEB-INF/jsp/header.jsp" flush="true" />

  <jsp:body>
      <form:form commandName="adminGroup" action="updateGroup.do">
          <form:errors path="*"></form:errors><br/>
          组名:<form:input path="groupName" readonly="true"/>
        <input type="submit" value="更新" name="update"/>
        <br/>

          <display:table name="adminGroup.rules" id="row">
              <display:column property="authorityName" title="权限"></display:column>
              <display:column title="查询">
                  <form:checkbox path="rules[${row_rowNum-1}].canRead"/>
              </display:column>
              <display:column title="更新">
                  <form:checkbox path="rules[${row_rowNum-1}].canWrite"/>
              </display:column>
          </display:table>
          描述:<br/>
          <form:textarea path="description" cols="80"/>
      </form:form>
      <br/>
    <a href="groupAdmin.do">返回组管理画面</a>
  </jsp:body>
</jsp:root>
```

除了在添加新组的视图部分强调的问题，我再追加一个，这个问题很容易让人觉得“丈二和尚摸不着头脑”。

问题的关键在于`<form:form>`的`action`属性，如果我们不明确指定该属性的话，表单将提交到最初Web请求发起的路径，这会造成什么问题呢？以我们当前更新组的Web请求为例，在`MultiAcitonController`重定向Web请求之后，发送到`GroupUpdateController`处理的Web请求路径对应是.../simplefx/updateGroup.do?groupName=AdminGroupOne。表单显示的时候，`GroupUpdateController`抽取参数绑定到Command对象，然后在视图中，`groupName`的值将被赋予`<form:input path="groupName" readonly="true"/>`，到这里没有任何问题。现在用户提交表单，如果我们没有明确指定action属性指向何处，那么，表单默认提交到发起请求的路径，也就是.../simplefx/updateGroup.do?groupName=AdminGroupOne。这样，路径中存在一个groupName参数，表单中也存在一个groupName参数（`<form:input path="groupName" readonly="true"/>`也将以groupName为参数名被提交)。数据绑定的时候，`CroupUpdateController`将发现当前请求中存在两个groupName参数，就会将它们当作一个`String`数组而绑定给Command对象的groupName属性，结果可想而知。我们最终要处理的Command对象的groupName可能变成了“AdminGroupOne,AdminGroupOne”。而实际

上，应该为“AdminGroupOne”。至于解决方法，当然是重新明确指定一下action对应的提交路径。

至此SimpleFormController的旅程已到终点了，不知你是否已经完全掌握该类的秉性了呢？（什么？还没有？那只能自己再查阅更多相关资料咯！）

24.2.4　AbstractWizardFormController

在使用Eclipse或者Netbeans之类的IDE进行开发的时候，如果我们新建一个工程或者文件，这些IDE通常都会提供多步的向导帮助我们一步一步完成相应部件的创建。在某些Web应用程序中，也存在类似的操作场景，比如，要注册某个网站的会员，注册过程可能就包括多步，每一步会提示输入某一方面的信息，以帮助我们简化操作流程。对于这种向导式的简单的多页面流程实现，我们可以求助于AbstractWizardFormController，它可以帮助我们简化类似场景的开发工作。

> **注意**　实际上，AbstractWizardFormController也只是面向简单的多页面流程处理，它所管理的多个页面表单中的数据，最终都是绑定到一个Command对象上的。只不过是每个页面绑定一部分而已。如果要在Web应用程序中处理更复杂的页面流程交互，建议集成Spring Web Flow（http:// www.springframework.org/Webflow），*Expert Spring MVC and Web Flow*一书对Spring Web Flow进行了详尽的介绍。

1. AbstractWizardFormController浅析

说白了，AbstractWizardFormController实现类就是要处理组成向导流程的所有页面所发起的Web请求。但与SimpleFormController处理单个表单页面不同，AbstractWizardFormController要管理多个表单页面的显示以及提交数据的处理工作。

图24-12对AbstractWizardFormController如何管理整个向导流程给出了一个概况，我们对AbstractWizardFormController的认识将在这个的基础上展开。

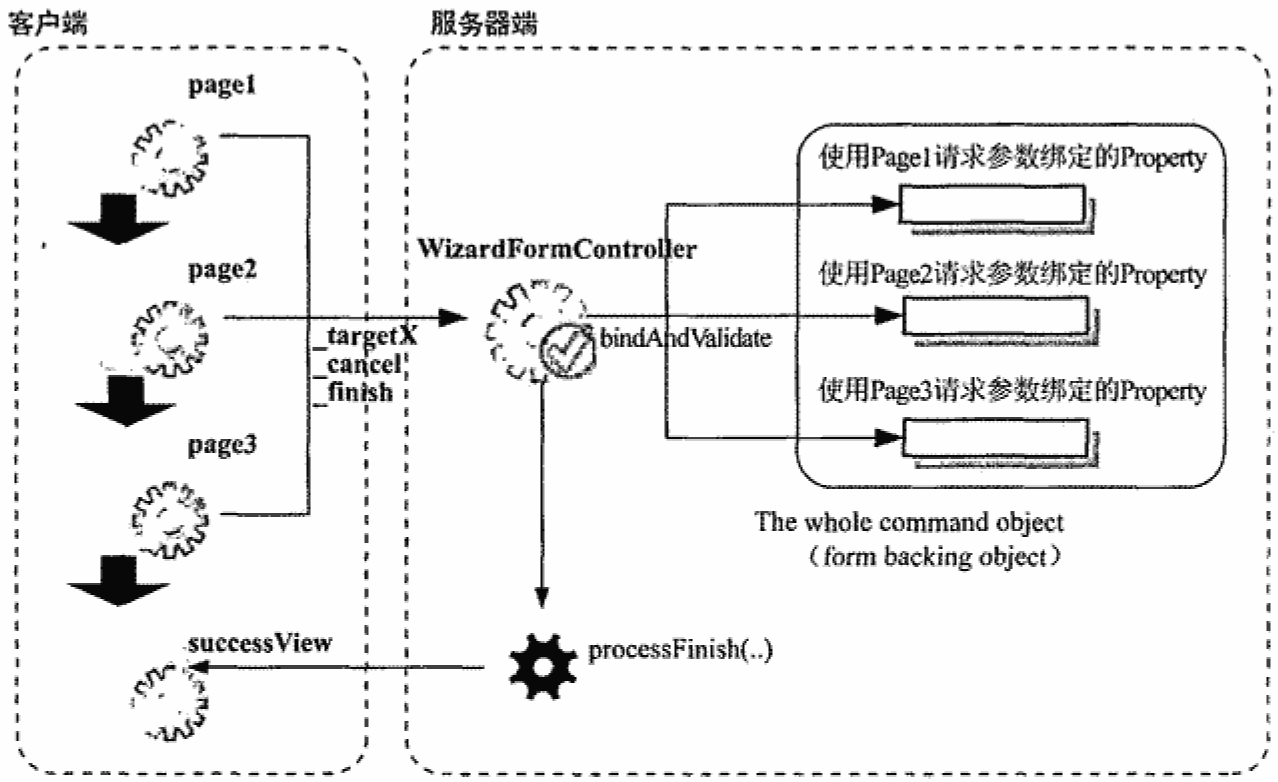

图24-12　AbstractWizardFormController的Web请求处理结构

Page1、Page2和Page3是组成向导流程的页面，这些页面提交的Web请求将统一由我们的AbstractWizardFormController接收并处理。但AbstractWizardFormController如何知道当前该显示哪个页面？它又是如何判断哪个页面提交的请求是最终的结束向导流程的Web请求呢？AbstractWizardFormController将根据每个页面请求所包含的特定参数来决定如何对当前请求进行处理，这些特定的参数我们可以分为如下三类。

- **PARAM_TARGET参数**。PARAM_TARGET参数用于指定目标页面，它的形式为_target后缀页面索引，比如_target0、_target1等。当AbstractWizardFormController实现类接收到这种类型的参数的时候，它只是将当前请求中的相应参数绑定到Command对象上，然后根据PARAM_TARGET参数后缀的目标页面索引显示相应的页面。
- **PARAM_FINISH参数**。PARAM_FINISH参数的表示形式为_finish。如果AbstractWizard-FormController实现类接收到该参数，则表示整个向导流程结束，可以调用process-Finish(..)方法处理最终的表单数据，并将页面转向任何一个想要转到的页面。
- **PARAM_CANCEL参数**。PARAM_CANCEL参数的表示形式为_cancel。当接收到该参数的时候，表示用户要取消当前的向导流程。默认情况下，AbstractWizardFormController不支持该参数的处理，但我们可以通过覆写processCancel(..)方法来改变这种默认行为。唯一要做的，可能只是返回一个ModelAndView实例，其中只包含了一个逻辑视图名。

实际上，AbstractWizardFormController实现类本质上依然是像SimpleFormController那样，分两个阶段来管理表单页面的处理，只不过是从逻辑上将单个表单页面划分为了多个表单页面，而最终绑定数据的Command对象却只有一个。AbstractWizardFormController将根据_targetX参数决定显示表单页面的某一部分，对应到视图那就是显示哪个向导页面。在该向导页面提交请求之后，AbstractWizardFormController会把它显示并提交的那部分数据绑定到唯一的那个Command对象上，只有在所有向导页面都提交之后，Command对象的数据才算绑定完成。从这样的角度来说，AbstractWizardFormController只不过是将一部分信息划分为多步来显示和处理罢了。

我们在向导（Wizard）最终结束的画面提交_finish参数之后，AbstractWizardFormController将从Command对象获取到所有向导页面（Wizard Page）搜集来的数据，然后它就可以调用process-Finish(..)方法开始“压轴戏”了。实际上，在实现AbstractWizardFormController的时候，也只有processFinish(..)方法是需要我们必须去实现的。

我想，以上的分析内容已经足以说明AbstractWizardFormController的底细。不过，如果你还意犹未尽的话，不妨再研读一下*Expert Spring MVC and Web Flow*一书。像如何提供数据验证，如何覆写某些方法以添加扩展逻辑之类的细节，该书都有提及。相信我，因为我已经读过了。

2. AbstractWizardFormController实例

我搜刮了FX系统的前台和后台的所有场景，也没有找到一个需要向导的流程。所以，为了演示AbstractWizardFormController的使用，我也不能免俗，创造了一个调查问卷的场景。我们将分三步引导用户完成对购车意愿的问卷调查。当然，整个问卷调查看起来有些简单了，但如果需要，我们也可以在这个基础上提供更加友好、更加全面的问卷调查向导。

为了分步显示调查的内容，我们提供了4个页面分别显示整体的某一部分内容，它们的关系如下所述。

- welcome.jsp。问卷调查向导的开始页面，提供简单的说明信息，如图24-13所示。
- carManufacturers.jsp。该页面将提供汽车厂商信息列表供用户选择，如图24-14所示。

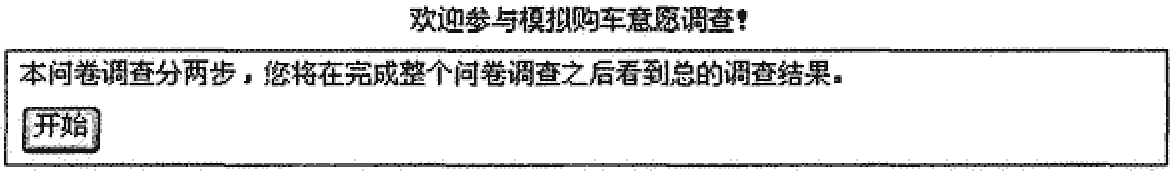

图24-13 webcome.jsp页面

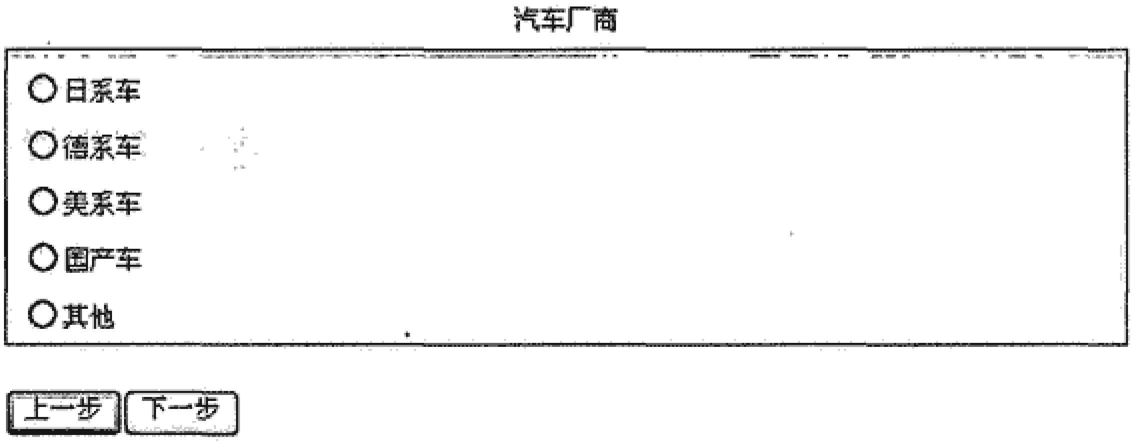

图24-14 carManufacturers.jsp页面

- carPriceScope.jsp。向导流程中最后一个问卷调查页面，显示意愿购车价格区间，点击"结束问卷"提交按钮即结束当前问卷调查，见图24-15。

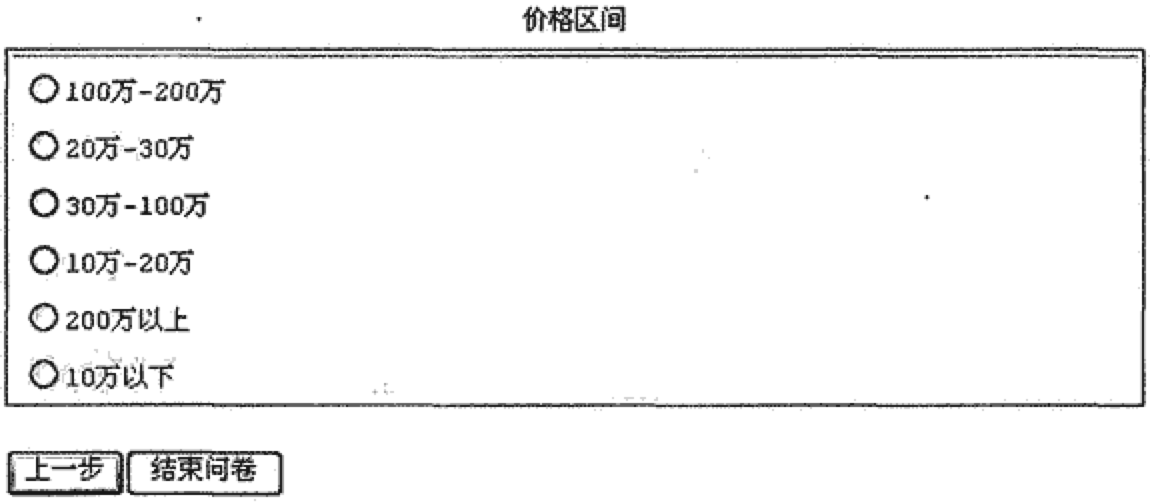

图24-15 carPriceScope.jsp页面

- surveyResult.jsp。确切地讲，该页面不能算向导的一部分，它只是调查结束后的一个结果显示页面，对应的是图24-12中的successView。当然，将具体数据转换为柱状图显示或许更讨人爱一些，该页面最终效果如图24-16所示。

现在让我们着手实现该向导流程。

- **购车意愿问卷调查功能实现流程概述**

(1) 构建业务层支持。

我们首先定义了ICarSurveyService接口作为业务层的对外窗口，如下所示：

```
public interface ICarSurveyService {
```

```
    List<String> getAvailableCarManufacturers();
    List<String> getAvailableCarPriceScope();
    SurveyResult addSurvyForSum(CarTrend carTrend);
}
```

汽车厂商调查结果

原产地	结果
日系车	11
德系车	13
美系车	13
国产车	10
其他	3

价格区间调查结果

价格区间	结果
100万-200万	23
20万-30万	21
30万-100万	34
10万-20万	19
200万以上	8
10万以下	10

图24-16 surveyResult.jsp页面

前两个方法将分别返回可供用户选择的调查选项，最后一个方法会将当前用户的问卷调查结果添加到系统中，然后返回一个目前为止总的调查结果。当然，要让ICarSurveyService帮忙，我们依然需要给出它的一个实现类。不过，因为这个MockObject实现要添加模拟数据，所以看起来太长。这里就不做罗列了，感兴趣可以猜猜是怎么玩的，反正实际系统中，我们不可能捏造数据，大都是从数据库中抽取，对吧？不管怎么样，在有了ICarSurveyService的实现类之后，一定不要忘了添加到顶层WebApplicationContext中，以便相应的Controller能够获得它的鼎力支持。

CarTrend类是我们将使用的Command对象类型（当然，我们并非让“作为Command对象”成为它存在的唯一目的），其两个属性（property）将分别保存当前用户提交的不同类型的问卷结果。该类的最终定义如代码清单24-24所示。

代码清单24-24 CarTrend类定义代码

```
public class CarTrend {
    private String carFrom;
    private String carPriceScope;

    public String getCarFrom() {
        return carFrom;
    }
    public void setCarFrom(String carFrom) {
        this.carFrom = carFrom;
    }
    public String getCarPriceScope() {
        return carPriceScope;
```

```
    }
    public void setCarPriceScope(String carPriceScope) {
        this.carPriceScope = carPriceScope;
    }
}
```

SurveyResult只用来保存每种问卷调查项以及每项的最终结果数据，其定义如代码清单24-25所示。

代码清单24-25 SurveyResult类定义代码

```
public class SurveyResult {
    private List<CarManufacturerStatus> carManufacturerStatus;
    private List<CarPriceScopeStatus>   carPriceScopeSatus;

    public SurveyResult()
    {
        carManufacturerStatus = new ArrayList<CarManufacturerStatus>();
        carPriceScopeSatus = new ArrayList<CarPriceScopeStatus>();
    }

    public List<CarManufacturerStatus> getCarManufacturerStatus() {
        return carManufacturerStatus;
    }
    public void setCarManufacturerStatus(List<CarManufacturerStatus> ➥
    carManufacturerStatus) {
        this.carManufacturerStatus = carManufacturerStatus;
    }
    public List<CarPriceScopeStatus> getCarPriceScopeSatus() {
        return carPriceScopeSatus;
    }
    public void setCarPriceScopeSatus(List<CarPriceScopeStatus> carPriceScopeSatus) {
        this.carPriceScopeSatus = carPriceScopeSatus;
    }
}
```

对于SurveyResult中涉及的两个值对象，我想就不用浪费篇幅了，它们只是保持每项的名称和结果值而已。

(2) 构建AbstractWizardFormController实现类。

主角出场时间，代码清单24-26给出的是AbstractWizardFormController具体实现类。

代码清单24-26 CarTrendSurveyWizardController类定义代码

```
public class CarTrendSurveyWizardController extends AbstractWizardFormController {
private ICarSurveyService carSurveyService;
private String successView;

public CarTrendSurveyWizardController()
{
    setCommandClass(CarTrend.class);
}

@Override
protected Map referenceData(HttpServletRequest request, int page) ➥
throws Exception {
```

```
    Map rData = new HashMap();
    switch(page)
    {
    case 1:
        List<String> carManufacturers = ➥
        getCarSurveyService().getAvailableCarManufacturers();
        rData.put("carManufacturers", carManufacturers);
        return rData;
    case 2:
        List<String> carPriceScopes = getCarSurveyService().getAvailableCarPriceScope();
        rData.put("carPriceScopes", carPriceScopes);
        return rData;
    default:
        return null;
    }
}

@Override
protected ModelAndView processFinish(HttpServletRequest arg0,
    HttpServletResponse arg1, Object command, BindException arg3) ➥
    throws Exception {
        CarTrend currentCarTrend = (CarTrend)command;
        SurveyResult sr = getCarSurveyService().addSurvyForSum(currentCarTrend);
        ModelAndView mav = new ModelAndView(getSuccessView());
        mav.addObject("surveyResult", sr);
        return mav;
}
// getter和setter方法定义
}
```

对于"提供依赖的服务对象"和"在构造函数中指定Command对象的类型"这些固定动作，估计你都熟知了，无需再提。successView是我们特地添加的一个属性，用于提供问卷结束后应该导向的视图名称，不要将它与SimpleFormController的successView混淆。SimpleFormController已经为我们定义了这么一个属性，而AbstractWizardFormController没有，所以，要用，我们就得自己定义。

如果对于单个表单页面来说，要为其提供附加显示数据，当然是通过referenceData(..)方法进行啦。比较幸运的是，在AbstractWizardFormController实现类中，我们也是通过覆写referenceData(..)方法做同样的工作。只不过，该referenceData(..)方法要多一个page参数，用于标识返回的数据到底要给哪个向导页面使用，谁让我们现在要管理多个页面的显示呢！

> **注意** 页面的索引是以0开始的，尽量遵循的是数组索引的"传统习俗"，_targetX参数也是一个道理，_target0指的是向导的第一个页面，_target1指的是第二个页面，依次类推。

实现一个AbstractWizardFormController必须要提供的覆写方法就是processFinish(..)。我们在该方法中调用服务对象将当前的调查结果告知系统，然后将返回的总的问卷调查情况，通过ModelAndView公开给下一个页面，也就是successView对应的最终结果显示页面。

以下是我们的CarTrendSurveyWizardController添加到特定于DispatcherServlet的

WebApplicationContext的情景：

```
<bean name="/survey.do" ➥
class="cn.spring21.simplefx.controllers.CarTrendSurveyWizardController">
    <property name="carSurveyService" ref="carSurveyService"/>
    <property name="successView" value="surveyResult"/>
    <property name="pages" value="welcome,carManufacturers,carPriceScope"/>
</bean>
```

AbstractWizardFormController需要知道所有组成向导流程的页面，所以，我们得通过pages属性告知一下。

> **注意** pages属性为String[]类型，所以，在通过value指定多个视图名的时候，我们采用逗号分隔多个视图名。为了清晰，也可以采用嵌套的<list>进行相同效果的表达。

(3) 构建视图模板。

有关视图模板，我想只展示中间两个的代码就能够说明问题了，如代码清单24-27所示。

代码清单24-27　部分试图模板代码示例

```
# /WEB-INF/jsp/carManufacturers.jsp
<jsp:root version="1.2" xmlns:jsp="http:// java.sun.com/JSP/Page"
    xmlns:display="urn:jsptld:http:// displaytag.sf.net"
    xmlns:c="urn:jsptld:http:// java.sun.com/jsp/jstl/core"
    xmlns:form="urn:jsptld:http:// www.springframework.org/tags/form">
  <jsp:directive.page contentType="text/html; charset=UTF8" />
  <jsp:include page="/WEB-INF/jsp/header.jsp" flush="true" />

  <jsp:body>
      <form:form>
        <display:table name="carManufacturers" id="carManufacturer">
            <display:caption>汽车厂商</display:caption>
            <display:column>
                <form:radiobutton path="carFrom" label="${carManufacturer}" ➥
                value="${carManufacturer}"/>
            </display:column>
        </display:table>
          <input type="submit" name="_target0" value="上一步"/>
          <input type="submit" name="_target2" value="下一步"/>
      </form:form>
  </jsp:body>
</jsp:root>

# /WEB-INF/jsp/carPriceScope.jsp
<jsp:root version="1.2" xmlns:jsp="http:// java.sun.com/JSP/Page"
    xmlns:display="urn:jsptld:http:// displaytag.sf.net"
    xmlns:c="urn:jsptld:http:// java.sun.com/jsp/jstl/core"
    xmlns:form="urn:jsptld:http:// www.springframework.org/tags/form">
  <jsp:directive.page contentType="text/html; charset=UTF8" />
  <jsp:include page="/WEB-INF/jsp/header.jsp" flush="true" />

  <jsp:body>
      <form:form>
          <display:table name="carPriceScopes" id="scope">
```

```
            <display:caption>价格区间</display:caption>
            <display:column>
                <form:radiobutton path="carPriceScope" value="${scope}" ➥
                label="${scope}"/>
            </display:column>
        </display:table>
        <input type="submit" name="_target1" value="上一步"/>
        <input type="submit" name="_finish" value="结束问卷"/>
    </form:form>
  </jsp:body>
</jsp:root>
```

其实，主要是想让你看看_target参数和_finish参数的样子。至于如何显示页面，其实可以因人、因项目、因太多原因而变化。

24.2.5 其他可用的 Controller 实现

除了我们已经了解的两大门派的几位主要弟子，Spring MVC的Controller体系下还有几个用于不同目的的Controller实现类，如下所述。

AbstractCommandController。AbstractCommandController是规范操作派里面的少数派。它继承了BaseCommandController的数据绑定以及数据验证的功能支持，自身也定义了简单的请求处理流程，但它的主要目的不是面向表单的处理（该方面的功能已经由AbstractFormController家族接管了）。实际上，我们可以认为AbstractCommandController只是一个加强型的AbstractController。直接继承AbstractController与直接继承AbstractCommandController差不多，唯一的区别可能就是，后者进一步规范了参数的获取和验证操作，从如下的代码摘录中就可以看出这一点：

```
protected ModelAndView handleRequestInternal(HttpServletRequest request, ➥
HttpServletResponse response) throws Exception {
    Object command = getCommand(request);
    ServletRequestDataBinder binder = bindAndValidate(request, command);
    BindException errors = new BindException(binder.getBindingResult());
    return handle(request, response, command, errors);
}
```

继承AbstractCommandController，只需要实现handle(..)模板方法即可，因为之前数据绑定和验证都已经搞定了。这就好比继承AbstractController，我们只负责实现一个handleRequestInternal(..)模板方法，而像页面缓存的设置等功能，则由AbstractController这个父类帮我们搞定。

总之，通过继承AbstractCommandController实现一个Controller，我们只需要关心数据绑定和验证之后的处理逻辑的实现即可。

ParameterizableViewController。ParameterizableViewController的实现逻辑十分简单，或者说干脆就没有处理逻辑，只是返回一个只包含指定逻辑视图名的ModelAndView实例，其请求处理方法的代码如下所示：

```
protected ModelAndView handleRequestInternal(HttpServletRequest request, ➥
HttpServletResponse response) throws Exception {
    return new ModelAndView(getViewName());
}
```

该Controller实现类的主要目的是为了帮助我们将“对具体视图的直接请求”纳入到Spring MVC

的统一处理体系之下，这样，我们依然可以依赖于ViewResolver的映射能力向客户端屏蔽不同视图类型的差异性。

比如，如果我们在页面内只是想通过超链接（HyperLink）访问某个页面（比如/WEB-INF/jsp/help/Help4sth.jsp），期间不需要任何处理，那么我们就可以在特定于DispatcherServlet的WebApplicationContext中添加如下Controller定义：

```
<bean name="/simpleFileRequest.do" ➥
class="org.springframework.Web.servlet.mvc.ParameterizableViewController">
    <property name="viewName" value="help/Help4sth"></property>
</bean>
```

通过viewName指定具体的视图名，ParameterizableViewController将不做任何处理，直接将对应“simpleFileRequest.do”的请求导向指定的视图。

UrlFilenameViewController。使用ParameterizableViewController一次只能映射一个视图文件。如果有一组视图文件都需要不做任何处理直接返回的话，我们就得使用UrlFilenameViewController。比如，大部分系统中可能会有专用的系统帮助信息，这些帮助信息大都实现为静态信息页面，客户端对这些帮助信息页面的请求将不需要做任何预处理。假设我们的帮助页面都存放在/WEB-INF/jsp/help/目录下，为了免去一个页面配置一个处理Controller之苦，我们就可以在特定于DipatcherServlet的WebApplicationContext中添加一个UrlFilenameViewController的定义，让它全权管理对帮助页面的访问，如下所示：

```
<bean name="/help/*" class="org.springframework.Web.servlet.mvc.UrlFilenameViewController">
    <property name="prefix" value="help/"></property>
</bean>
```

客户端通过如下链接方式访问，即可访问到需要的帮助页面（对应如下链接的是/WEB-INF/jsp/help/Help4sth.jsp）：

```
<a href="<c:url value='help/Help4sth.do'/>">Get Help</a>
```

我们可以通过UrlFilenameViewController的prefix和suffix属性进一步修饰UrlFilenameViewController的映射能力，有关这些内容可以参考该类的Javadoc文档。

ServletForwardingController和ServletWrappingController。ServletForwardingController是将对当前Controller的请求转发（Forward）给当前应用中定义的某个Servlet，也就是说，将Controller与Servlet的地位拉到了同一个水平。而ServletWrappingController则是将当前应用中的某个Servlet直接包装为一个Controller，所有到ServletWrappingController的请求实际上是由它内部所包装的这个Servlet来处理的。这两种Controller实现类更多的是为了集成现有的Servlet，比如将包含某些现有逻辑的Servlet纳入到Spring MVC的处理体系中。我们可以在ServletForwardingController和ServletWrappingController的Javadoc文档中获得使用它们的详细信息。

> **注意**　除了AbstractCommandController希望并且也必须继承使用之外，其他几个Controller实现都不是以继承为目的而存在的，只需要相应的配置即可使用。

基本上，经典意义上的Controller实现类也就这些了。下面是Controller返回的ModelAndView出场时间。

24.3 ModelAndView

通常，Controller在将Web请求处理完成后，会返回一个ModelAndView实例[1]。该ModelAndView实例将包含两部分内容，一部分为视图相关内容，可以是逻辑视图名称，也可以是具体的View实例；另一部分则是模型数据，视图渲染过程中将会把这些模型数据合并入最终的视图输出。所以，简单来说，ModelAndView实际上就是一个数据对象。不过通过该数据对象，我们却可以解除具体的Web请求处理Controller与视图渲染之间的紧密耦合，使得两个方面能够独立演化。

为了方便实例化ModelAndView，该类定义了两组参数各异的构造方法，一组使用逻辑视图名称（logic named view）标志视图，一组直接使用View实例标志视图，如下所示：

```
public ModelAndView(String viewName)
public ModelAndView(String viewName,Map model)
public ModelAndView(String viewName,String modelName,Object modelObject)

public ModelAndView(View view)
public ModelAndView(View view,Map model)
public ModelAndView(View view,String modelName,Object modelObject)
```

每组的第一个构造方法只接受视图信息，所以，构造完成后，我们得通过addAllObject(..)或者addObject(..)实例方法，向构造完成的ModelAndView实例添加模型数据；每组第二个构造方法则可以同时指定视图信息和模型数据信息，一步到位；如果要添加到模型的只有一个数据对象，那么可以使用每组的第三个构造方法，该构造方法属于第二个构造方法的简化版。除了以上的构造方法之外，ModelAndView还有一个默认的没有参数的构造方法，如果使用该构造方法实例化对象，那么之后就需要使用其他实例方法来设置视图和模型数据信息了（听起来有点儿像废话哦）。

24.3.1 ModelAndView 中的视图信息

恰如两组构造方法所表明的那样，ModelAndView可以以逻辑视图名的形式或者View实例的形式来保存视图信息。如果ModelAndView中直接返回了具体的View实例，那么，DispatcherServlet将直接从ModelAndView中获取该View实例并渲染视图，如下所示：

```
View view = null;
...
view = mv.getView();
view.render(mv.getModelInternal(), request, response);
```

否则（当然是返回逻辑视图名称啦），DispatcherServlet将寻求ViewResolver的帮助，根据ModelAndView中的逻辑试图名称获取一个可用的View实例，然后再渲染视图，如下所示：

```
View view = null;
...
view = resolveViewName(mv.getViewName(), mv.getModelInternal(), locale, request);
view.render(mv.getModelInternal(), request, response);
```

注意 虽然通过ModelAndView可以保存视图的逻辑名称或者具体的View实现类，但是我们更倾向于使用逻辑视图名来标志视图。这样可以给我们的视图选择带来很大的灵活性，除非必要，尽量不要直接返回具体的View实例。

---

① 也可能返回null，表示Controller内部自行处理视图的渲染。

24.3.2　ModelAndView 中的模型数据

ModelAndView以org.springframework.ui.ModelMap的形式来保持模型数据，通过构造方法传入的或者通过实例方法添加的模型数据都将添加到这个ModelMap中。至于ModelMap中保持的模型数据将会在视图渲染阶段，由具体的View实现类来获取并使用。

我们需要为添加到ModelAndView的一组或者多组模型数据提供相应的键(Key)，以便具体的View实现类可以根据这些键获取具体的模型数据，然后公开给视图模板。通常，模型中的数据对应的键需要与视图模板中的标志符相对应，如图24-17所示。

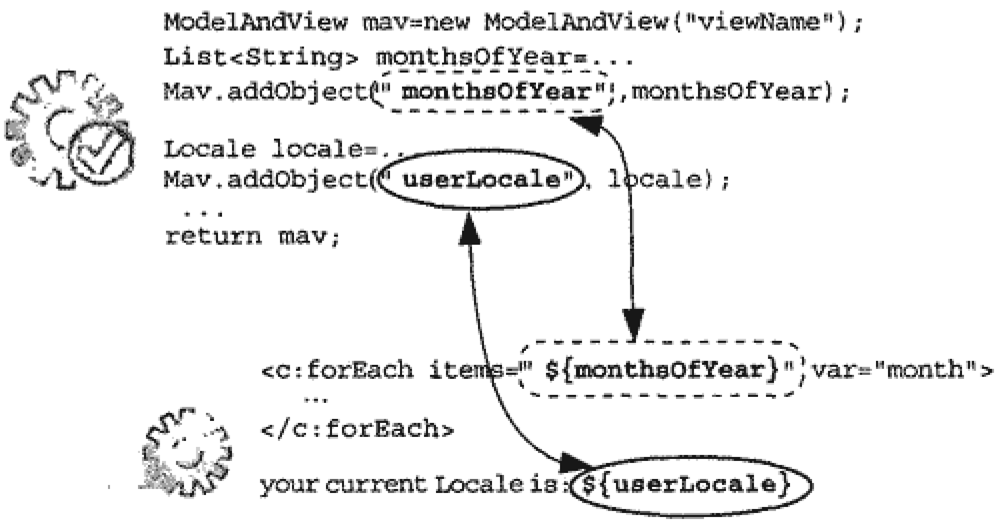

图24-17　模型数据与视图模板的对应关系

基于JSP/JSTL模板的视图实现，通常是将模型数据通过HttpServletRequest的属性（Attribute）的形式公开给具体的模板。而像基于Velocity之类的通用模板引擎的视图实现，则会将ModelAndView中的模型数据复制到它们自己的数据获取上下文中，比如Velocity的Context。但不管什么视图类型，对应的视图模板都将可以通过添加到ModelAndView的模型数据的键来获取模型数据，并合并到最终的视图输出结果中。

24.4　视图定位器 ViewResolver

我们已经知道了ViewResolver的主要职责是，根据Controller所返回的ModelAndView中的逻辑视图名，为DispatcherServlet返回一个可用的View实例。现在是揭开ViewResolver如何“尽职”的时候了。

有ViewResolver的职责为前提，理解甚至于自己声明一个ViewResolver接口变得不再困难。实际上ViewResolver接口定义确实很简单，如下所示：

```
public interface ViewResolver {
    View resolveViewName(String viewName, Locale locale) throws Exception;
}
```

接口实现类只需要根据resolveViewName()方法中以参数形式传入的逻辑视图名（viewName）和当前Locale的值，返回相应的View实例即可[1]。至于每个ViewResolver实现类如何处理具体

① 同时传入Locale的目的是，在需要的情况下，可以根据Locale的不同返回不同的视图实例，这对于支持国际化的视图是必要的。

的逻辑视图名与具体的View实例之间的对应关系，则因实现类的不同而存在差异。

大部分的ViewResolver实现类，除了org.springframework.web.servlet.view.BeanNameViewResolver是直接实现ViewResolver接口，都直接或者间接继承自org.springframework.web.servlet.view.AbstractCachingViewResolver。因为针对每次请求都重新实例化View将可能为Web应用程序带来性能上的损失，所以Spring MVC在AbstractCachingViewResolver这一继承层次加入了View实例的缓存功能。AbstractCachingViewResolver默认启用View的缓存功能。对于生产环境来说，这是合理的默认值。不过，如果在测试或者开发环境下，我们想即刻反映相应的修改结果，可以通过setCache(false)暂时关闭AbstractCachingViewResolver的缓存功能。

Spring MVC在AbstractCachingViewResolver的基础上为我们提供了一系列的ViewResolver实现。下面让我们来认识一下它们的庐山真面目。

24.4.1 可用的 ViewResolver 实现类

为了便于理解，我们可以将Spring MVC提供的ViewResolver划分为两类，一类称为“面向单一视图类型的ViewResolver，另一类则称为面向多视图类型的ViewResolver。下面是这两类ViewResolver的详细情况。

1. 面向单一视图类型的ViewResolver

该类别ViewResolver的正宗名称应该是UrlBasedViewResolver（它们都直接地或者间接地继承自该类）。使用该类别的ViewResolver，我们不需要为它们配置具体的逻辑视图名到具体View的映射关系。通常只要指定一下视图模板所在的位置，这些ViewResolver就会按照逻辑视图名，抓取相应的模板文件、构造对应的View实例并返回。之所有又将它们称之为面向单一视图类型的ViewResolver，是因为该类别中，每个具体的ViewResolver实现都只负责一种View类型的映射，ViewResolver与View之间的关系是一比一。比如，我们之前一直使用的InternalResourceViewResolver，它通常就只负责到指定位置抓取JSP模板文件，并构造InternalResourceView类型的View实例并返回。而VelocityViewResolver则只关心指定位置的Velocity模板文件（.vm），并会将逻辑视图名映射到视图模板的文件名，然后构造VelocityView类型的View实例返回，诸如此类。

属于该类别的主要ViewResolver实现类为如下几个。

- **InternalResourceViewResolver**。它是我们使用最多的ViewResolver实现类型，它对应InternalResourceView[①]视图类型的映射，说白了也就是处理JSP模板类型的视图映射。如果DispatcherServlet在初始化的时候，不能在自己的WebApplicationContext中找到至少一个ViewResolver，那么，InternalResourceViewResolver将作为默认的ViewResolver被使用。
- **FreeMarkerViewResolver/VelocityViewResolver**。FreeMarkerViewResolver和VelocityViewResolver分别负责对应FreeMarkerView和VelocityView类型视图的查找工作，它们将根据逻辑视图名到指定的位置获取对应的模板文件，并构造FreeMarkerView和VelocityView的实例返回给DispatcherServlet使用。
- **JasperReportsViewResolver**。JasperReportsViewResolver只关心根据逻辑视图名到指定位置查找JasperReport类型模板文件，并返回AbstractJasperReportsView的具体子类型View实例，例如JasperReportsCsvView或者JasperReportsHtmlView等。
- **XsltViewResolver**。只负责根据逻辑视图名查找并返回XsltView类型的View实例。

① 通常，jsp文件都是Web容器内部处理的，所以，internal resource的名字可能因此而来吧！

启用以上这些ViewResolver，与使用InternalResourceViewResolver一样简单。最基本的方法是，使用prefix属性指定模板所在路径，使用suffix属性指定模板文件的后缀名。这样，在获取逻辑视图名之后，相应的ViewResolver内部就能够根据[prefix]+viewName+[suffix]这样的URL找到对应的模板文件，并构造对应的View实例而返回了。以VelocityViewResolver的使用为例，至于其他的几个ViewResolver的使用，你基本上就可以“举一反三”了，更加详尽的配置项，可以参考对应类的Javadoc或者*Professional Java Development with the Spring Framework*一书中对应视图章节的介绍内容。下面给出了针对VelocityViewResolver的配置代码示例：

```
<bean id="viewResolver" ➥
class="org.springframework.Web.servlet.view.velocity.VelocityViewResolver">
    <property name="prefix" value="../velocity/"/>
    <property name="suffix" value=".vm"/>
</bean>
```

现在DispatcherServlet对视图的请求将会由VelocityViewResolver接管，VelocityViewResolver将根据传入的逻辑视图名，到指定目录下查找.vm类型的Velocity模板文件，并构造VelocityView实例返回给DispatcherServlet使用。就跟我们所说的那样，它只负责到指定位置查找对应Velocity的单一视图类型，而不会返回其他，比如Freemarker视图对应的View实例。

> **注意**　关于使用Velocity作为视图技术需要附加的配置内容，可以参考稍后ResourceBundleViewResolver部分的附带信息。

2. 面向多视图类型的ViewResolver

使用面向单一视图类型的ViewResolver，我们不需要指定明确的逻辑视图名与具体视图之间的映射关系，对应的ViewResolver将自动到指定位置匹配自己所管辖的那种视图模板，并构造具体的View实例。面向多视图类型的ViewResolver则不然。使用面向多视图类型的ViewResolver，我们需要通过某种配置方式明确指定逻辑视图名与具体视图之间的映射关系，这可能带来配置上的烦琐。不过，好处是，面向多视图类型的ViewResolver可以顾及多种视图类型的映射管理。如果你的逻辑视图名想要映射到InternalResourceView，那么面向多视图类型的ViewResolver可以做到。如果你的逻辑视图名想要映射到VelocityView，那么，面向多视图类型的ViewResolver也可以做到。相对于只支持单一视图类型映射的情况，面向多视图类型的ViewResolver更加灵活。

面向多视图类型的ViewResolver的主要实现类有三个，它们分别是ResourceBundleViewResolver、XmlViewResolver以及BeanNameViewResolver。以下是它们的详细情况介绍。

ResourceBundleViewResolver。ResourceBundleViewResolver构建在ResourceBundle上，继承了ResourceBundle国际化支持的能力，也是所有的ViewResolver实现类中唯一提供视图国际化支持的ViewResolver。ResourceBundleViewResolver管理的视图的逻辑名称与具体视图的映射关系保存在properties文件中，格式符合Spring的IoC容器的properties配置格式。ResourceBundleViewResolver内部将通过PropertiesBeanDefinitionReader加载这些配置信息。之后，根据逻辑视图名查找的操作，实际上也就简化为beanfactory.getBean(viewName)的形式了（当然，实际上要做的事情会多一些）。

使用ResourceBundleViewResolver之前，我们得先将其添加到DispatcherServlet的WebApplicationContext中，如下所示：

```
<bean id="resourceBundleViewResolver" ➥
```

```
class="org.springframework.Web.servlet.view.ResourceBundleViewResolver">
</bean>
```

如果我们没有指定properties配置文件从何处加载的话，ResourceBundleViewResolver默认将从classpath的根路径加载以**views**为basename的properties文件，比如views.properties、views_zh_CN.properties等。如果我们想改变这种默认加载行为，可以通过setBasename (String)或者setBasenames(String[])方法来进行变更。

以下是一个典型的ResourceBundleViewResolver使用的properties配置文件内容：

```
viewTemplate.class=org.springframework.Web.servlet.view.InternalResourceView
viewTemplate.(abstract)=true

help/HelpForSomething.(parent)=viewTemplate
help/HelpForSomething.url=/WEB-INF/jsp/help/HelpForSomething.jsp

hello.class=org.springframework.Web.servlet.view.velocity.VelocityView
hello.url=cn/spring21/simplefx/resources/velocity/hello.vm
# 其他视图定义……
```

视图的bean定义主要有两个属性：class和url。如果我们想要避免每次为同一类型的视图指定某些共同的属性，也可以定义一个模板声明，然后通过parent引用该模板声明。这些特性在第二部分中我们已经领教过了，不是吗？

注意　如果要在ResourceBundleViewResolver中使用Velocity或者Freemarker之类的通用模板引擎渲染的视图，那么需要在WebApplicationContext中添加相应的配置，使得视图渲染阶段能够获取模板引擎的支持。实际上，单独使用VelocityViewResolver或者Freemarker-ViewResolver也需要同样的配置。

我们以使用Velocity类型视图的配置为例，（Freemarker类型视图的配置与Velocity类型视图的配置雷同）。在应用程序的WebApplicationContext中，我们添加org.springframework.web.servlet.view. velocity.VelocityConfigurer的配置如下：

```
<bean id="velocityConfig" ➥
class="org.springframework.Web.servlet.view.velocity.VelocityConfigurer">
    <property name="configLocation"
        value="/WEB-INF/velocity-config.properties"/>
/bean>
```

这样，在视图渲染阶段就可以根据该配置获取一个VelocityEngine进行视图模板与数据的合并（Merge）操作，以便最终输出视图页面。

velocity-config.properties的配置内容，完全就是特定于Velocity的内容了。你可以参考Velocity的相关文档获取配置参数，这里可以给出一个简单的实例，如下所示：

```
resource.loader = classpath
classpath.resource.loader.description = Classpath Resource Loader
classpath.resource.loader.class = ➥
org.apache.velocity.runtime.resource.loader.ClasspathResourceLoader
classpath.resource.loader.path=.

velocimacro.library =
```

最后，对于Velocity（或者Freemarker）的模板文件，最好像我们给出的配置内容所指定的那样，将它们放入应用程序的classpath中进行加载，而不是依赖于默认的文件系统加载行为。

XmlViewResolver。XmlViewResolver与ResourceBundleViewResolver之间最主要的区别就是，它们所采用的配置文件格式不同。ResourceBundleViewResolver按照Spring IoC容器所接受的properties配置格式配置逻辑视图名与具体视图之间的映射关系，而XmlViewResolver则是按照Spring IoC容器接受的XML配置文件格式来加载映射信息。与ResourceBundleViewResolver同样的配置信息，使用XmlViewResolver的话，内容如代码清单24-28所示。

代码清单24-28　XmlViewResolver使用配置代码示例

```
<bean  name="viewTemplate" ➥
class="org.springframework.Web.servlet.view.InternalResourceView" ➥
abstract="true">
</bean>

<bean name="help/HelpForSomething" parent="viewTemplate">
    <property name="url" value="/WEB-INF/jsp/help/HelpForSomething.jsp"/>
</bean>

<bean  name="hello" ➥
class="org.springframework.Web.servlet.view.velocity.VelocityView" ➥
p:url="cn/spring21/simplefx/resources/velocity/hello.vm">
</bean>
```

XmlViewResolver默认会加载/WEB-INF/views.xml作为配置文件。不过，我们可以在将XmlViewResolver添加到WebApplicationContext的时候，根据情况改变这一默认行为，例如：

```
<bean id="xmlViewResolver" class="org.springframework.Web.servlet.view.XmlViewResolver">
    <property name="location" value="classpath:views.xml"/>
</bean>
```

现在，XmlViewResolver将从Classpath的根路径加载名为views.xml的配置文件。至于其他配置，比如Velocity需要的VelocityConfigurer，因为与使用何种ViewResolver没有关系，只与是否使用Velocity作为视图技术有关，所以依然需要根据情况添加到WebApplicationContext中。

> **注意**　*XmlViewResolver并不支持视图的国际化（I18n）。如果必须对国际化视图给予支持，需要使用ResourceBundleViewResolver。*

BeanNameViewResolver。BeanNameViewResolver可以认为是XmlViewResolver的原型版或者简化版。使用它，我们可以直接将View实例注册到当前DispatcherServlet所使用的特定的WebApplicationContext中，而不用像XmlViewResolver那样另辟一块地。不过，BeanNameViewResolver更多地用于快速搭建应用框架原型，或者构建小型的Web应用程序。对于正常的基于Spring MVC的Web应用程序，应尽量避免将可以分离出来的视图配置信息一并加入到DispatcherServlet的WebApplicationContext中。

至于如何启用BeanNameViewResolver作为ViewResolver，我想你现在要比我清楚，如下所示：

```
<bean id="beanNameViewResolver" ➥
class="org.springframework.Web.servlet.view.BeanNameViewResolver"/>
```

至于具体视图的配置，参照XmlViewResolver即可。

实际上，正如我们所看到的那样，这三种ViewResolver在本质上是一样的，只不过是配置的表

现形式上存在差异而已。最终的配置信息都将转换为Spring IoC容器中管理的View实例，BeanName-ViewResolver应该是最初的实现原型吧！

24.4.2 ViewResolver 查找序列（Chain Of ViewResolver）

虽然我们在之前的示例中一直都是使用一个InternalResourceViewResolver进行视图查找，但这并不意味着每个基于Spring MVC的Web应用程序只能使用一个ViewResolver。实际上，DispatcherServlet不但可以接受多个HandlerMapping以处理Web请求到具体Handler的映射，也可以接受多个ViewResolver以处理视图的查找。

DispatcherServlet初始化时，将根据类型扫描自己的WebApplicationContext中定义的ViewResolver。如果查找到存在多个ViewResolver的定义，DispatcherServlet将根据这些ViewResolver的优先级进行排序，然后当需要根据逻辑视图名查找具体的View实例的时候，将按照排序后的顺序遍历这些ViewResolver，只要期间任何一个ViewResolver返回非空的View实例，当前查找即告结束。如果DispatcherServlet没能在当前的WebApplicationContext中找到任何的ViewResolver定义，它将使用InternalResourceViewResolver作为默认的ViewResolver使用。

ViewResolver的优先级的指定使用Ordered接口作为标准，这已经成为Spring框架内设定优先级方式的惯例了。假设我们希望主要使用ResourceBundleViewResolver进行逻辑视图名到具体View实例的查找，如果没能找到，再寻求InternalResourceViewResolver的帮助。我们可以在DispatcherServlet的WebApplicationContext中添加如下配置内容：

```
<bean id="resourceBundleViewResolver" ➥
class="org.springframework.Web.servlet.view.ResourceBundleViewResolver">
    <property name="order" value="1"></property>
</bean>

<bean id="viewResolver" ➥
class="org.springframework.Web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/jsp/"/>
    <property name="suffix" value=".jsp"/>
</bean>
```

相应ViewResolver的bean定义对应的id或者name属性值是任意的，DispatcherServlet将按照类型来获取ViewResolver。如果没有为某个ViewResolver指定order值的话，默认值为Integer.MAX_VALUE，对应的是最低优先级。

如果为DispatcherServlet指定多个ViewResolver的话，不要给予InternalResourceViewResolver以及其他UrlBasedViewResolver子类过高的优先级，因为这些ViewResolver即使找不到相应的视图，也不会返回null以给我们轮询下一个ViewResolver的机会，这样，我们所指定的其他ViewResolver实际上就形同虚设。合理的处理方式是，给予ResourceBundleViewResolver或者XmlViewResolver这种能够通过返回null以表明无法找到相应视图的ViewResolver较高的优先级，而只是将InternalResourceViewResolver（或者其他类似行为的ViewResolver）添加为最低优先级ViewResolver，以作为DispatcherServlet的后备查找对象。

24.5 各司其职的 View

org.springframework.web.servlet.View是Spring MVC中将原本可能存在于DispatcherServlet中的视图渲染逻辑得以剥离出来的关键组件。通过引入该策略抽象接口，我们可以极具灵活

性地支持各种视图渲染技术。

org.springframework.web.servlet.View的接口定义如下：

```
public interface View {
    String getContentType();
    void render(Map model, HttpServletRequest request, HttpServletResponse response) ➥
    throws Exception;
}
```

各种View实现类主要的职责就是在render(..)方法中实现最终的视图渲染工作，但这些对DispatcherServlet来说是透明的，DispatcherServlet只是直接接触ViewResolver所返回的View接口，获得相应引用后把视图渲染工作转交给返回的View实例即可。至于该View实例是什么类型，具体如何完成工作，DispatcherServlet是无须关心的。不过，对于我们来说，了解各个View实现类的实现原理，有助于我们更好地理解整个框架是如何运作的，并且，如果现有的View不能够满足我们的需要，我们也可以自定义一个需要的View实现类。

24.5.1　View 实现原理回顾

总地来说，当前绝大多数的视图渲染技术都是构建在模板的原理上。我们回想一下，这种基于模板的视图生成方式在我们的生活中到处可见。

- 厨师为了能够提供统一样式的蛋糕，会使用模子来制作，只要提供不同成分的面团，经过相同的模子压制，就能够获得统一样式却不同口味的蛋糕。厨师用的模子（可能木质也可能金属质地）是不是与我们提供的JSP文件相似？那不同成分的面团跟我们提供的不同的模型数据是否类似？
- 篆刻后的方印，只要蘸上不同颜色的印泥就能印出同一式样但不同颜色的印章图案。方印就是模板，不同的印泥就是要表现的数据，是否可以这么理解呢？

实际上，不管是生活中还是视图渲染过程中，只要使用模板这种模式，它们的工作原理就是一条路子下来的，如图24-18所示。

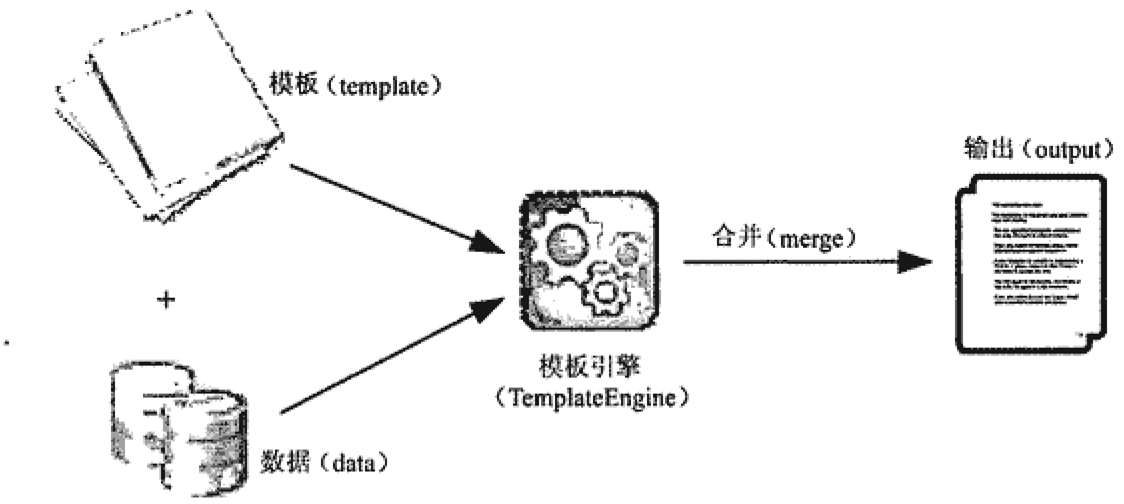

图24-18　使用模板的工作原理

所以，只要能够理解当前视图渲染的实现与生活中这些使用模板的场景之间的共同之处，那么，余下的工作将不再神秘。一个View实现类所要做的，就是使用相应的技术API将模板和最终提供的模型数据合并到一起，最终输出结果页面给客户端，所以，不难想象对应不同视图技术的View实现是一

个什么样子。

如果我们要使用JSP文件作为模板输出视图页面，那么我们的View实现类可能如代码清单24-29所示。

代码清单24-29 基于JSP的View实现类原型代码示例

```
public class JspView implements View {
    private String jspTemplateFileLocation;

    public String getContentType() {
    return "text/html;charset=UTF-8";
    }

    public void render(Map model, HttpServletRequest request, ➥
    HttpServletResponse response) throws Exception {
        response.setContentType(getContentType());
        exposeModelToRequest(model,request);
        request.getRequestDispatcher(jspTemplateFileLocation).forward(request, response);
        }

    protected void exposeModelToRequest(Map model,HttpServletRequest request)
    {
        if(!model.isEmpty())
        {
            Iterator<Map.Entry> iter = model.entrySet().iterator();
            while(iter.hasNext())
            {
                Map.Entry entry = iter.next();
                String attrName = (String)entry.getKey();
                Object attrValue = entry.getValue();
                request.setAttribute(attrName, attrValue);
            }
        }
    }
    // getter和setter方法定义……
}
```

JSP模板文件与模型数据的合并（merge）操作将由Web容器（比如Tomcat）来完成，所以，这里我们只是通过Servlet API将合并的工作转发给Web容器即可。

如果我们使用Velocity模板输出视图页面，那么我们的View实现类可能如代码清单24-30所示。

代码清单24-30 基于Velocity的View实现类原型代码示例

```
public class VelocityView implements View {
    private String vmTemplateLocation;
    private VelocityEngine engine;

    public String getContentType() {
        return "text/html;charset=UTF-8";
    }

    public void render(Map model, HttpServletRequest request, ➥
    HttpServletResponse response) throws Exception {
        response.setContentType(getContentType());
        Context ctx = new VelocityContext();
```

```
        copyMapToContext(model,ctx);
        engine.mergeTemplate(vmTemplateLocation, ctx, response.getWriter());
    }

    protected static void copyMapToContext(Map source,Context ctx)
    {
        Iterator iter = source.keySet().iterator();
        while(iter.hasNext())
        {
            String key = (String)iter.next();
            Object value = source.get(key);
            ctx.put(key, value);
        }
    }
    // getter和setter方法定义
}
```

如果我们要使用Excel作为输出对象，那么我们的View实现类可能如代码清单24-31所示。

代码清单24-31 基于Excel的View实现类原型代码示例

```
public class ExcelView implements View {
    private String xlsTemplateLocation;

    public String getContentType() {
        return "application/vnd.ms-excel";
    }

    public void render(Map model, HttpServletRequest request, ➥
    HttpServletResponse response) throws Exception {
        response.setContentType(getContentType());
        // 1.定位模板位置
        HSSFWorkbook workbook = readInExcelTemplate(xlsTemplateLocation);
        // 2.合并数据和模板
        mergeModelWithTemplate(model,workbook);
        // 3.输出到客户端
        ServletOutputStream out = response.getOutputStream();
        workbook.write(out);
        out.flush();
    }
    private void mergeModelWithTemplate(Map model, HSSFWorkbook workbook) { workbook.➥
    getSheetAt(1).getRow(11).getCell((short)1).setCellValue((String)model.get("dataKey"));
        // ...
    }

    protected HSSFWorkbook readInExcelTemplate(String location) throws Exception
    {
        File xlsFile = new File(location);
        InputStream ins = new FileInputStream(xlsFile);
        POIFSFileSystem fs = new POIFSFileSystem(ins);
        HSSFWorkbook workbook = new HSSFWorkbook(fs);
        return workbook;
    }
    // getter和setter方法定义
}
```

怎么样？虽然只是原型代码，但已经足够说明问题了，不是吗？实际上，Spring MVC提供的针对各种视图技术的View实现也是按照同一条路子走下来的，只不过比我们的原型代码要严谨罢了。

24.5.2 可用的 View 实现类

Spring MVC提供的View实现类都直接或者间接继承自org.springframework.web.servlet.view.AbstractView。该类定义了大多数View实现类都需要的一些属性和简单的模板化的实现流程。

AbstractView为所有View子类定义的属性是如下几个。

private String contentType = DEFAULT_CONTENT_TYPE。DEFAULT_CONTENT_TYPE的内容是“text/html;charset=ISO-8859-1”。我们可以通过contentType的setter方法更改这一默认值。

private String requestContextAttribute。requestContextAttribute属性是要公开给视图模板使用的org.springframework.web.servlet.support.RequestContext对应的属性名，比如，如果setRequestContextAttribute("rc")的话，那么，相应的RequestContext实例将以rc作为键放入模型中。这样，我们就可以在视图模板中通过rc引用到该RequestContext。通常情况下，如果我们使用Spring提供的自定义标签，那么不需要公开相应的RequestContext。但如果不使用Spring提供的自定义标签，那么为了能够访问处理过程中所返回的错误信息等，就需要通过公开给视图模板的RequestContext来进行了。可以参考RequestContext的Javadoc文档了解它能够赋予我们的能力。

private final Map staticAttributes = new HashMap()。如果视图有某些静态属性，比如页眉、页脚的固定信息等，只要将它们加入staticAttributes，那么，AbstractView将保证这些静态属性将一并放入模型数据中，最终一起公开给视图模板。既然所有的View实现子类都继承自AbstractView，那么它们也就都拥有了指定静态属性的能力。比如我们在“面向多视图类型支持的ViewResolver”中定义视图映射的时候，为某些具体视图定义指定了静态属性，如下所示：

```
<bean name="viewTemplate" ➥
class="org.springframework.Web.servlet.view.InternalResourceView" ➥
abstract="true" ➥
p:attributesCSV="copyRight= spring21.cn ,author= fujohnwang ">
</bean>
```

那么，现在我们就可以像普通的模型数据那样，在视图模板中访问这些静态属性，如下所示：

```
...
Author: ${author}
<br/>
Copyright: ${copyRight}
...
```

不过，除了通过attributesCSV属性以CSV字符串形式传入多个静态属性，我们还可以通过attributes属性以Properties的形式传入静态属性，或者通过attributesMap属性以Map的形式传入静态参数。

AbstractView除了定义了以上公共属性以外，还定义了一个简单的模板化的方法流程。

(1) 将添加的静态属性全部导入到现有的模型数据Map中，以便后继流程在合并视图模板的时候可以获取这些数据。

(2) 如果requestContextAttribute被设置（默认为null），则将其一并导入现有的模型数据Map中；

(3) 根据是否要产生下载内容，设置相应的HTTP Header。

(4) 公开renderMergedOutputModel(..)模板方法给子类实现。

这样，AbstractView的直接或者间接子类，就可以在现有属性和流程的基础上进行开发了。

AbstractView中一个主要的扩展类是org.springframework.web.servlet.view.AbstractUrlBasedView，AbstractUrlBasedView为子类提供的公共设施很简单，只有一个String型的url。那些需要根据模板路径读入模板文件的View实现，大都属于AbstractUrlBasedView门下。

AbstractView和AbstractUrlBasedView是所有View实现类的“总统领”，那些不需要指定url的View实现类大都归于AbstractView门下，余下的则由AbstractUrlBasedView管辖。在这样的前提下，我们再来看各种实际可用的View实现类。

1. 使用JSP技术的View实现

属于该类别的View实现主要包括：

- org.springframework.web.servlet.view.InternalResourceView
- org.springframework.web.servlet.view.JstlView
- org.springframework.web.servlet.view.tiles.TilesView
- org.springframework.web.servlet.view.tiles.TilesJstlView

其中，org.springframework.web.servlet.view.InternalResourceView是面向JSP技术的主要View实现类，它们之间的关系如图24-19所示。

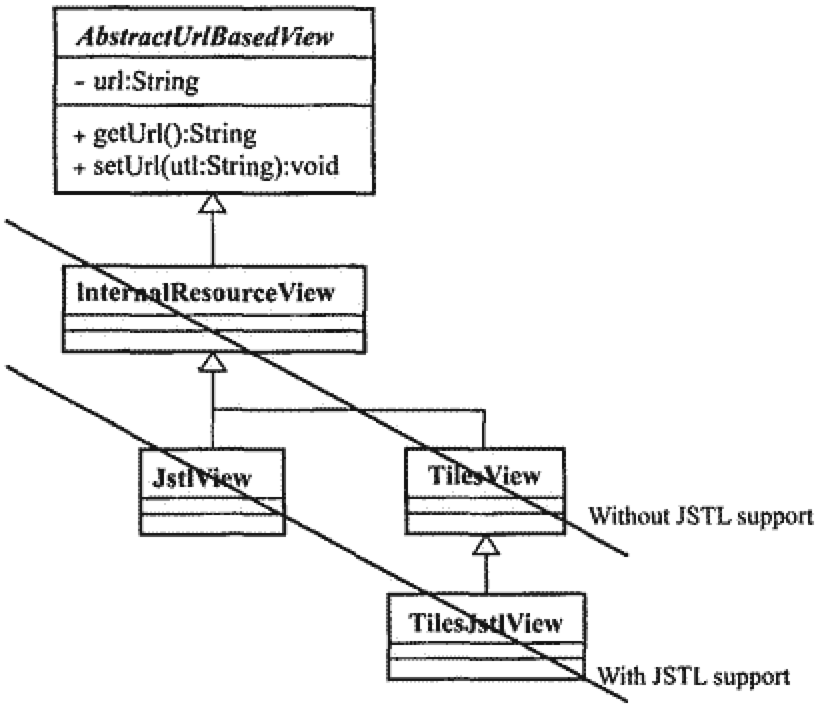

图24-19　使用JSP的View实现类

InteralResourceView和JstlView都是面向单一JSP模板的View实现，二者的区别在于J2EE 1.4之前的Web应用程序不支持JSTL。所以，这些Web应用程序只能使用InternalResourceView，而之后的Web应用程序因为支持JSTL，所以，使用JstlView是没有问题的。TilesView和TilesJstlView之间的区别与InteralResourceView和JstlView是类似的。不过，TilesView和TilesJstlView使用了Struts的Tiles视图技术，它们支持的是复合JSP视图。另外，Spring 2.5之后也引入了对Tiles 2（http://tiles.apache.org/）的支持，对应的TilesView实现位于org.springframework.web.servlet.view.tiles2包下面，与org.springframework.web.servlet.view.tiles包下面的Tiles 1.x版本的TilesView和TilesJstlView相区别。

这些使用JSP技术的View实现，虽然可以在“面向多视图类型的ViewResolver”的映射关系中单

独配置，不过，因为它们有特定于自己的ViewResolver，即InternalResourceViewResolver，所以，更多时候，只需要在使用之前变换一下如下配置中具体的viewClass类型即可：

```
<bean id="viewResolver" ➥
class="org.springframework.Web.servlet.view.InternalResourceViewResolver">
    <property name="viewClass" value="org.springframework.Web.servlet.view.JstlView"/>
    <property name="prefix" value="/WEB-INF/jsp/"/>
    <property name="suffix" value=".jsp"/>
</bean>
```

不过，Tiles视图的使用与单纯的JSP视图在使用上存在一点儿差异，我们需要为TilesView和TilesJstlView的渲染提供必需的DefinitionsFactory。这个工作可以通过TilesConfigurer类完成，将TilesConfigurer添加到WebApplicationContext之后，它将为容器内的TilesView和TilesJstlView的渲染提供绑定到ServletContext的DefinitionsFactory。TilesConfigurer的配置如下所示：

```
<bean id="tilesConfigurer" ➥
class="org.springframework.Web.servlet.view.tiles.TilesConfigurer">
   <property name="definitions">
     <list>
       <value>/WEB-INF/defs/tiles-def1.xml</value>
       <value>/WEB-INF/defs/tiles-def2.xml</value>
       ...
     </list>
   </property>
 </bean>
```

2. 使用通用模板技术的View实现

通用模板技术现在比较主流的是Velocity和Freemarker。如果我们的Web应用程序要启用这两种技术渲染视图，那么，Spring MVC提供了FreeMarkerView和VelocityView两种View实现。

因为二者都是基于同样的理念构建视图，所以，FreeMarkerView和VelocityView有着共同的父类AbstractTemplateView，它们之间的继承层次关系如图24-20所示。

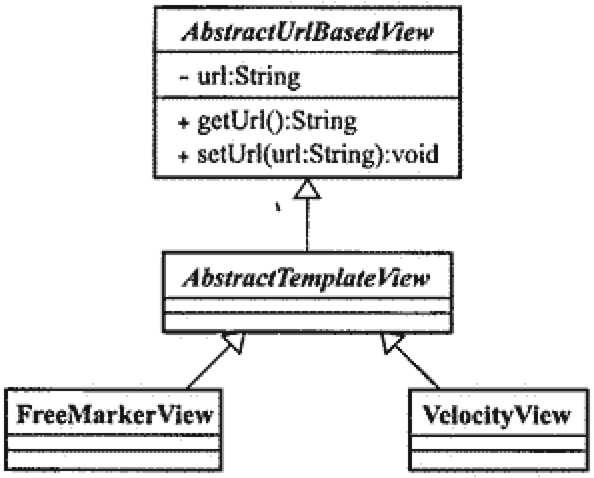

图24-20 使用通用模板技术的View实现

AbstractTemplateView定义了几个boolean属性，让我们可以决定是否公开暴露某些数据给最终的合并过程，如下所述。

- **private boolean exposeRequestAttributes = false**。是否需要将request中的所有属性公开给合并过程，默认为false。

- **private boolean allowRequestOverride = false**。是否允许request中的属性覆盖ModelAndView中同名的attribute，默认不允许这么做。
- **private boolean exposeSessionAttributes = false**。是否要将session中的属性公开给视图模板与模型数据的合并过程，默认不做。
- **private boolean allowSessionOverride = false**。是否允许session中同名的属性覆盖掉返回的ModelAndView中的属性，默认也是不允许这么做。
- **private boolean exposeSpringMacroHelpers = true**。是否需要为Spring提供的宏（macro）公开一个需要的RequestContext对象，默认需要，将以“springMacroRequestContext”为键公开一个RequestContext给合并过程。

除了这些，FreeMarkerView和VelocityView自身也定义了几个属性可以进一步限定视图渲染过程，比如VelocityView允许我们通过dateToolAttribute和numberToolAttribute公开Velocity Tools（http:// velocity.apache.org/tools/devel/）的DateTool和NumberTool给模板使用。

FreeMarkerView和VelocityView的使用都有相应的ViewResolver支持，即FreeMarkerViewResolver和VelocityViewResolver。不过，我们也可以在“面向多视图类型的ViewResolver”中使用它们。唯一需要注意的就是，使用这两种视图类型的时候，不要忘记通过FreeMarkerConfigurer和VelocityConfigurer为它们提供渲染过程中使用的模板引擎支持。

3. 面向二进制文档格式的View实现

该类的View实现主要指Excel和PDF形式的文档视图，通过设定合适的contentType，并且本地有相应的应用程序的话，这些文档将可以在浏览器中直接打开，而不是下载保存。

对于Excel形式的视图，Spring MVC提供了如下两个抽象类的视图实现。

- **AbstractExcelView**。使用Apache POI（http:// poi.apache.org/）来构建Excel文档的View实现类，支持读入Excel模板文件，子类需要实现buildExcelDocument(..)模板方法，以给出具体的模型数据到模板文件的合并逻辑。
- **AbstractJExcelView**。该抽象类使用JExcel API（http:// www.andykhan.com/jexcelapi/）作为视图的渲染API，同样支持现有Excel模板文件的读入，具体子类也需要通过实现buildExcelDocument(..)模板方法，来实现具体的模型数据到Excel模板文件的合并过程。

两种面向Excel的View实现类都支持按照Locale读入不同的Excel模板文件，读入顺序类似于：

(1) fileLocation_zh_CN.xls;

(2) fileLocation_zh.xls;

(3) fileLocation.xls。

也就是说，我们可以为不同地区的用户提供不同的视图文件。

对应PDF形式的View实现类只有AbstractPdfView，它将使用iText来构建最终要输出的PDF文件。应该是API的限制，该类无法读入PDF形式的模板文件（当然，没有API的支持，也不可能做到）。我们只能通过该类创建新的PDF文件，然后将模型数据与要输入的格式一并纳入新创建的PDF文件对象中。该类也是抽象类，子类要实现buildPdfDocument(..)模板方法提供具体的输出逻辑。

因为面向二进制文档格式的View实现没有一个统一的模板形式，所以，Spring MVC无法提供通用的View实现类，只能在抽象父类中提供部分共同逻辑的实现，而具体的模型数据如何融入视图的显示逻辑，则需要子类在相应的模板方法中给出。

有关面向二进制文档格式的View实现的使用，我们可能需要使用“面向多视图类型的ViewResolver”，因为没有特定于二进制文档格式View实现的ViewResolver可用。

4. 面向JsperReport的View实现

面向JsperReport的View实现允许我们输出JasperReport生成的相应格式的报表文件，包括HTML格式、CSV格式、Excel格式以及PDF格式。只要我们在ModelAndView中将要合并到报表的数据返回，面向JsperReport的View实现将把这些数据按照指定格式输出到客户端。

面向JsperReport的View实现主要包括如下几个。

- **AbstractJasperReportsSingleFormatView**。只负责输出单一类型的报表文件的View抽象类，实现了不同模板类型的读入以及数据的合并操作，将不同报表格式的输出通过模板方法下发给具体的子类实现，包括：
  - JasperReportsCsvView
  - JasperReportsHtmlView
  - JasperReportsPdfView
  - JasperReportsXlsView

  如果只需要根据模型数据输出单一文档格式的报表视图，选择以上对应的View子类即可。
- **JasperReportsMultiFormatView**。允许根据ModelAndView中的某个模型数据的值来决定输出何种格式的报表文档，默认使用“format”作为键。当然，我们可以通过setFormatKey(String)来更改这一默认键的名称。

如果在ModelAndView中添加如下数据，并且使用JasperReportsMultiFormatView作为将要使用的View实现：

```
ModelAndView mav = new ModelAndView(...);
mav.addObject("format","pdf");
...
return mav;
```

那么，JasperReportsMultiFormatView最终将通过JasperReportsPdfView输出PDF格式的报表文档。关于format的值与具体View实现类之间的关系，如表24-2所示。

表24-2 format的值与具体View实现类之间的关系

format值	具体View类型
csv	JasperReportsCsvView
html	JasperReportsHtmlView
pdf	JasperReportsPdfView
xls	JasperReportsXlsView

当然，我们可以通过setFormatMappings(Properties)方法更改这一默认映射行为。

至于面向JsperReport的View实现类的使用，我们既可以使用特定的JasperReportsViewResolver来映射逻辑视图名到具体View实现类，也可以使用ResourceBundleViewResolver之类“面向多视图类型的ViewResolver”。更多的信息可以参考Spring的参考文档，其中对各种视图的应用有详细的介绍，但对于JasperReport相关的View实现，主要设定的属性可能只有viewClass、url和reportDataKey，如下所示：

```
reportView.class=org.springframework.Web.servlet.view.jasperreports.JasperReportsPdfView
reportView.url=/WEB-INF/reports/jasperReportTemplate.jasper
reportView.reportDataKey=customJrDataSource
```

其中，reportDataKey作为ModelAndView中JasperReport需要的数据源（JRDataSource）的键，通

常是必须的。

5. 使用XSLT技术的View实现

我想或许Cocoon（http:// cocoon.apache.org/）才是使用XSLT技术实现视图的典型应用吧。不过，如果我们的应用程序也想以XSLT的形式处理数据的显示，Spring MVC也为我们提供了相应的View实现。

要在基于Spring MVC的Web应用程序中使用XSLT处理数据到视图的显示，有如下两种View实现供选择。

AbstractXsltView。使用AbstractXsltView进行XSLT数据转换，需要我们实现一个AbstractXsltView的子类，在该子类中覆盖模板方法Source createXsltSource(..)，在该模板方法中将Model中的数据转换为javax.xml.transform.Source类型返回，如代码清单24-32所示。

代码清单24-32　AbstractXsltView具体实现类代码示例

```
public class SimpleXsltView extends AbstractXsltView {
    @Override
    protected Source createXsltSource(Map model, String root, ➥
    HttpServletRequest request, HttpServletResponse response) throws Exception {
        Document document = ➥
        DocumentBuilderFactory.newInstance().newDocumentBuilder().newDocument();
        Element rootElement = document.createElement(root);
        String author = (String)model.get("Author");
        Element authorNode = document.createElement("author");
        authorNode.setTextContent(author);
        rootElement.appendChild(authorNode);
        return new DOMSource(rootElement);
    }
}
```

然后只需要配置该视图给相应的ViewResolver即可，比如：

```
simpleXslt.class=..SimpleXsltView
simpleXslt.stylesheetLocation=/WEB-INF/xsl/simpleStyleSheet.xslt
```

XsltView。使用XsltView要比AbstractXsltView省却不少麻烦，起码我们不用重新实现一个子类，我们只需要在ModelAndView中返回指定键标志的javax.xml.transform.Source实例即可，如下所示：

```
ModelAndView mav = new ModelAndView(..);
Source source = ...;
mav.addObject("xmlSource",source);
...
return mav;
```

其中“xmlSource”是XsltView寻找数据源时候要使用的键，需要在配置视图的时候指定：

```
xsltViewDemo.class=org.springframework.Web.servlet.view.xslt.XsltView
xsltViewDemo.url=/WEB-INF/xsl/simpleStyleSheet.xslt
xsltViewDemo.sourceKey=xmlSource
```

实际上，使用这两种View的主要工作应该在于XSLT式样表文件的编写，至于View的使用，只不过是配置一下罢了。不过，不管怎么样，除非你的应用程序之前有一致的要求，使用XSLT终归会强制你提供特定类型的数据，慎用。

6. RedirectView和逻辑视图名前缀

org.springframework.web.servlet.view.RedirectView所做的工作，与以上所有提到的View

实现类有点儿"大不同啊大不同"，就跟它的名字所展示的那样，RedirectView会对指定的Url做重定向操作。

如果设置RedirectView的http10Compatible属性为true，RedirectView将直接通过HttpServletReponse的sendRedirect(..)方法进行重定向操作，否则通过设置HTTP状态码（303）以及HTTP Header（"Location"）达到同样的目的。不过在此之前，RedirectView会将ModelAndView中的模型数据附加到指定的URL后部，然后对URL进行编码。

使用RedirectView最多的地方是在Controller内，当然，通过相应的ViewResolver指定也是可以的，例如：

```
ModelAndView mav = new ModelAndView();
RedirectView view = new RedirectView("addUser.do");
mav.setView(view);
...
return mav;
```

不过，不管是在Controller内直接实例化RedirectView使用，还是在相应的ViewResolver中配置，看起来都不是很简洁的样子。所以，Spring MVC还提供了另外一种进行请求重定向的方法，那就是在逻辑视图名中使用redirect或者forword前缀。

实际上，我们在前面已经接触过这两个字符前缀的使用了。如果在刚才的代码中使用结合redirect前缀的逻辑视图名，代码看起来就是如下所示：

```
ModelAndView mav = new ModelAndView("redirect:addUser.do");
...
return mav;
```

当然，我们可能更愿意将逻辑视图名以注入的方式取得，这在最初的实例中我们已经领教过了。

24.5.3 自定义 View 实现

虽然Spring MVC框架已经提供了足够多的View实现类支持，但有些情况下依然无法满足我们的应用要求。不过，好在框架自身对视图渲染相关关注点的分离，使得扩展并添加自定义View实现类也不是什么难事。

我们要添加的自定义View实现类所处的场景是这样的：FX系统中的交易等各类信息需要以报表的形式提供给相应顾客以及后台管理员，所以，在用户前台画面和后台管理画面上定义有相应的链接。点击这些链接之后，需要返回相应的PDF或者CSV格式的报表文件进行显示。需求实际上很简单，不过Spring MVC提供的JasperReport相关的View实现类，以及面向二进制文档之类的View实现类（包括AbstractPdfView、AbstractExcelView等），都属于那种根据Web请求实时生成报表文件并输出的逻辑范畴。对于某些系统来说，这样的处理是合适的，但FX系统的报表根据法律要求，需要保存3～5年的时间。这些报表将是法律依据，每个顾客的信息，每笔数据的信息都不能遗漏。所以，在FX系统中，这些报表是通过批处理（Batch）在某一个时段全部输出的，输出后的信息保持于数据库中。当前台画面或者后台画面请求相应报表文件的时候，只需要根据Web请求信息到数据库中获取具体的文件名，并将对应的文件输出到客户端即可。自定义的View实现类所要做的，只是将已经通过批处理输出的PDF或者CSV报表文件传输给客户端显示。

我们可以继承AbstractView类。不过，既然同样需要根据URL去获取文件，直接继承AbstractUrlBasedView或许更合适一些。当然，这完全视具体情况而定。代码清单24-33是我们完成的自定义View实现类StaticPdfView的代码演示。

代码清单24-33　用于静态PDF文件输出的自定义View实现类代码示例

```
public class StaticPdfView extends AbstractUrlBasedView {

    public StaticPdfView()
    {
        setContentType("application/pdf");
    }

    @Override
    protected void renderMergedOutputModel(Map model, HttpServletRequest request, ➥
    HttpServletResponse response) throws Exception {
        response.setContentType(getContentType());
        InputStream ins = new FileInputStream(getUrl());
        OutputStream out = response.getOutputStream();
        IOUtils.copy(ins, out);
        out.flush();
        IOUtils.closeQuietly(ins);
        IOUtils.closeQuietly(out);
    }
}
```

十分简单，不是吗？只是根据url信息读取文件，然后通过response输出到客户端即可。而View实现类里面所要做的不就是这些吗？根据具体场景构建视图内容，然后通过HttpServletResponse的Writer或者OutputStream，将构建后的视图内容输出到客户端。就这点儿工作！

因为通常的ViewResolver实现都继承了AbstractViewResolver的默认开启缓存功能，所以，通过ViewResolver来查找并使用我们的StaticPdfView并非合适的方式。另外，当前场景中，StaticPdfView所需要的url信息是从数据库获取的，显然无法通过相应ViewResolver进行定义（当然，不排除其他场景下结合ViewResolver使用StaticPdfView的情形）。鉴于以上两点，在相应的Controller内部直接构造StaticPdfView的实例并返回，或许是比较合适的做法。也就是说，在Controller中是直接返回View实例还是返回逻辑视图名，需要根据情况权衡。在大多数情况下，推荐使用返回逻辑视图名的做法，但不排除直接返回View实例的情况。StaticPdfUrlViewController提供了一段代码示例，演示了StaticPdfView的使用，如代码清单24-34所示。

代码清单24-34　Controller内StaticPdfView的使用代码示例

```
public class StaticPdfUrlViewController extends AbstractController {

    @Override
    protected ModelAndView handleRequestInternal(HttpServletRequest request, ➥
    HttpServletResponse response) throws Exception {
        ModelAndView mav = new ModelAndView();
        View view = constructView(request);
        mav.setView(view);
        mav.addObject(..,..); // 如果必要，添加其他模型数据
        return mav;
    }

    private View constructView(HttpServletRequest request) {
        String pdfFilePath = getPdfFileLocation(request);
        StaticPdfView view = new StaticPdfView();
        view.setUrl(pdfFilePath);
        return view;
    }
```

```
    protected String getPdfFileLocation(HttpServletRequest request){
        // 根据请求参数从数据库中取得相应的url信息
        String url = ...;

        return url;
    }
}
```

方法getPdfFileLocation(..)将抽取Web请求参数，然后根据这些参数查询数据库，并返回对应的PDF文件路径。具体实现可能随前后台之间的参数约定而有少许差异。

注意 通常，自定义View实现类需要结合相应的ViewResolver才能使用，直接在Controller中实例化View并非大部分情况下的做法。对某类View来说，完全可以为其单独声明一个ViewResolver，指定合适的优先级别（通过order属性）。即使现用的ViewResolver无法满足需要，为某类View实现类提供自定义的ViewResolver实现类也并非难事。各位不妨考虑下，类似StaticPdfView这样的View实现，除了在Controller中直接实例化，是否可以为其提供一个自定义的ViewResolver呢？

现在你是否准备自定义自己的View实现类了呢？

24.6 小结

HandlerMapping、Controller、ModelAndView、ViewResolver和View可以算是Spring MVC框架中的“五虎将”，它们共同组成了Spring MVC框架的强大躯干。本章对它们进行了详细的介绍，希望你完成本章内容之后，对它们已经了然于心了。不过，这五个角色并非Spring MVC的全部，没有了其他角色的支持，Spring MVC也不会看起来这么饱满。接下来，我们将一起看一下Spring MVC家族中的其他成员。

第25章 认识更多Spring MVC家族成员

本章内容

- 文件上传与MultipartResolver
- Handler与HandlerAdaptor
- 框架内处理流程拦截与HandlerInterceptor
- 框架内的异常处理与HandlerExceptionResolver
- 国际化视图与LocalResolver
- 主题（Theme）与ThemeResolver

在深入讲述Spring MVC框架之前，我们先暂时跳出对框架内主要角色的认知范围，再次“鸟瞰”Spring MVC框架总体上的逻辑结构。

到目前为止，我们主要认识了Spring MVC框架的五大主要角色，它们是HandlerMapping、Controller、ModelAndView、ViewResolver和View。在DispatcherServlet处理Web请求的过程中，它们顺序承担了相应的职责。我想，在之前的内容基础上，我们应该能够对整个Web请求的处理流程中各个角色所处的位置达成以下共识，如图25-1所示。

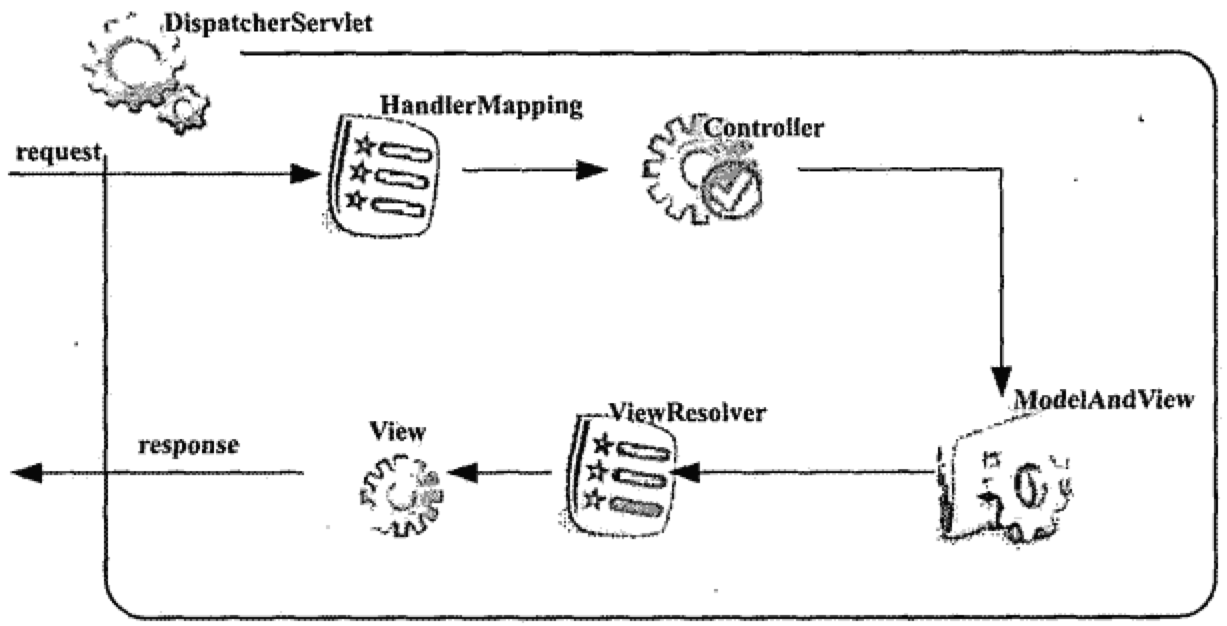

图25-1　三万英尺鸟瞰Spring MVC处理流程的逻辑结构

它们就好像Spring MVC的“骨架”，有了它们，即使整个框架看起来就像是个“骷髅兵”，在某种程度上已经足俱战斗力。为了能够让整个Spring MVC框架看起来更加饱满，我们还有一段路程要走。在此之前，我们可以先提前看一下地图，以免迷失方向（见图25-2）。

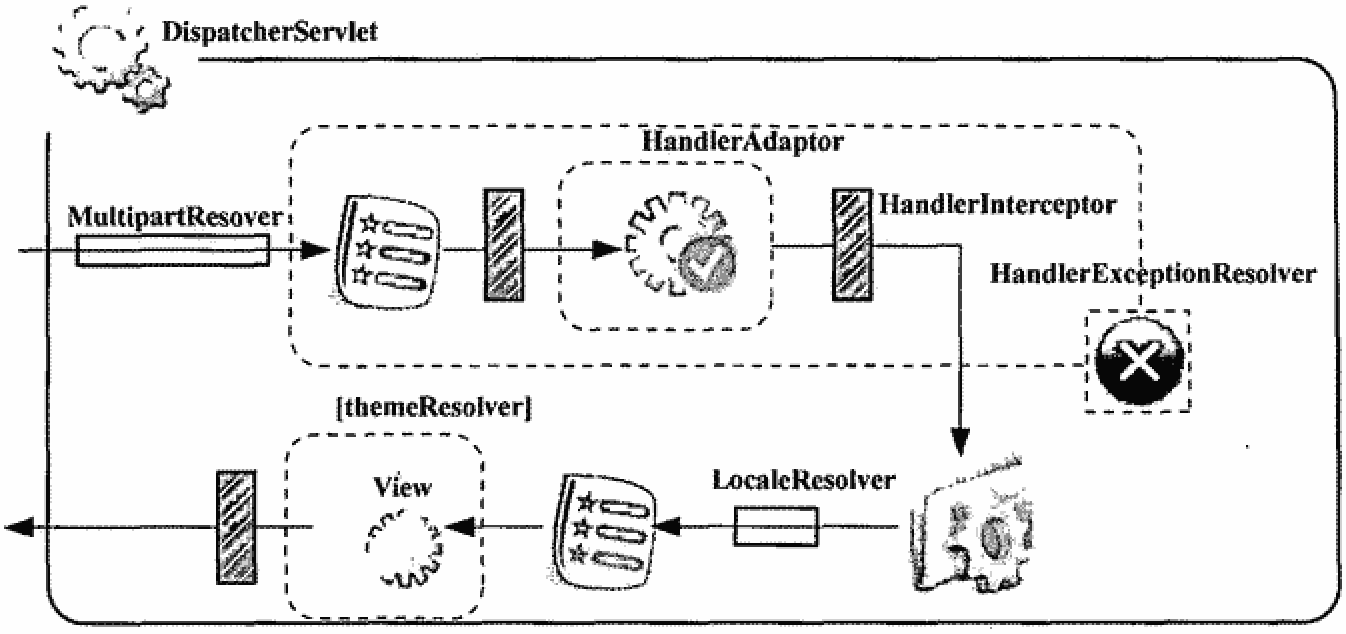

图25-2 几千米高空再次鸟瞰Spring MVC处理流程的逻辑结构

在这幅“地图”（图25-2）中，已经走过的“地点”只有图标，没有文字说明，而我们要经过的新的“地名”则都有标注。

- **MultipartResolver**。我们将经过的第一站，它位于HandlerMapping之前，简单来说，如果有文件上传的请求，它将会大展身手。
- **HandlerInterceptor**。HandlerInterceptor将对处理流程进行拦截。拦截的位置可以有三个地方可以选择，我想在“地图”中不难找到这些位置（斜线背景的竖向方框所标志的位置）。
- **HandlerAdaptor**。实际上，Spring MVC并不只是支持Controller一种Handler类型。HandlerAdaptor可以帮助我们使用其他类型的Handler。
- **HandlerExceptionResolver**。在处理具体Web请求的过程中，相应的Handler出现异常情况怎么办？HandlerExceptionResolver将为我们提供一种框架内的标准处理方式。
- **LocaleResolver**。有了LocaleResolver，根据用户的Locale显示不同的视图变得很容易。
- **ThemeResolver**。用户可以选择不同的主题（Theme）？ThemeResolver正是为这而生的！

下面让我按照顺序，带大家逐一领略“地图”中每一地点的“风土人情”。在按照“地图”的指示完成整个旅程的时候，我们将能够在开发过程中完全驾驭整个Spring MVC框架，不信？走着瞧啊！

25.1 文件上传与 MultipartResolver

如果要在基于Spring MVC的Web应用程序中通过表单上传文件，那么MultipartResolver将是在服务器端处理文件上传的主要组件。

HTML页面中的表单最初所采用的application/x-www-form-urlencoded编码方式，并不足以满足文件上传的需要，所以，RFC 1867（http://www.faqs.org/rfcs/rfc1867.html）在此基础上增加了新的multipart/ form-data编码方式以支持基于表单的文件上传。通常情况下，按照如下形式声明表单以及表单中的元素：

```
<FORM action=".." method="post" enctype="multipart/form-data">
    <INPUT NAME="fileElement" TYPE="file">
    <INPUT TYPE="submit" VALUE="Upload">
</FORM>
```

客户端浏览器将按照RFC1867所规定的格式，对提交表单内容进行编码，服务器端只需要根据RFC1867规定的格式对请求中的信息进行解码，就可获得客户端表单提交的数据，包括上传的文件。

既然RFC1867所规定的规则是一定的，所以，我们没有必要每次都根据这一规则分析每一请求中的信息。既然是通用的逻辑，当然也就有通用的类库，比如早期的jsp smart upload和Oreilly的COS类库，以及现在使用最多的Commons FileUpload（http://commons.apache.org/fileupload/）类库。实际开发中，我们只需要使用这些专门针对基于表单的文件上传处理类库即可。

在实现基于表单的文件上传功能的时候，Spring MVC框架底层实际上也是使用了以上几种类库。只不过，通过org.springframework.web.multipart.MultipartResolver接口的抽象，Spring MVC将具体选用哪一种类库的权利留给了我们。

25.1.1　使用 MultipartResolver 进行文件上传的简单分析

MultipartResolver的接口定义如下：

```
public interface MultipartResolver {
    boolean isMultipart(HttpServletRequest request);
    MultipartHttpServletRequest resolveMultipart(HttpServletRequest request) throws ➥
    MultipartException;
    void cleanupMultipart(MultipartHttpServletRequest request);
}
```

当Web请求到达DispatcherServlet并等待处理的时候，DispatcherServlet首先会检查能否从自己的WebApplicationContext中找到一个名称为multipartResolver（由DispatcherServlet的常量MULTIPART_RESOLVER_BEAN_NAME所决定）的MultipartResolver实例。如果能够获得一个MultipartResolver的实例，DispatcherServlet将通过MultipartResolver的isMultipart(request)方法检查当前Web请求是否为multipart类型。如果是，DispatcherServlet将调用MultipartResolver的resolveMultipart(request)方法，并返回一个MultipartHttpServletRequest供后继处理流程使用，否则，直接返回最初的HttpServletRequest。

当Web请求类型为multipart的时候，MultipartResolver的resolveMultipart(request)所返回的MultipartHttpServletRequest将被作为后继处理流程所依赖的HttpServletRequest而使用。也就是说，对应最初请求的HttpServletRequest将被“偷梁换柱”为MultipartHttpServletRequest，此后处理流程各个环节中所使用的HttpServletRequest的具体类型为MultipartHttpServletRequest。当然，如果MultipartHttpServletRequest不能够提供比HttpServletRequest更多的能力，那么在这里“劳师动众”地使用Decerator模式进行“偷梁换柱”看起来也没太大意义了。下面让我们看该接口的定义：

```
public interface MultipartHttpServletRequest ➥
extends HttpServletRequest, MultipartRequest {
}

public interface MultipartRequest {
    Iterator getFileNames();
    MultipartFile getFile(String name);
```

```
    Map getFileMap();
}
```

MultipartHttpServletRequest的附加能力来自于它的父接口MultipartRequest。简单地说，我们现在可以在某个Controller中，通过MultipartHttpServletRequest直接获取MultipartFile类所封装的上传后的文件，如下所示：

```
MultipartHttpServletRequest request = (MultipartHttpServletRequest)originalRequest;
MultipartFile file = request.getFile("fileParameter");
...
```

至于要将MultipartFile中保存的上传文件存入数据库，还是写入文件系统，那就看个人喜好或者应用场景的需要了。

当然，接口永远是接口，具体工作还得有人来做（这话是不是说过好几遍了）。Spring MVC框架内为MultipartResolver提供了两个可用的实现类，即org.springframework.web.multipart.commons.CommonsMultipartResolver和org.springframework.web.multipart.cos.CosMultipartResolver。前者使用Commons FileUpload类库实现，后者则使用Oreilly Cos类库实现。要启用Spring MVC框架内的文件上传支持，本质上讲，就是选择这两个实现类中的哪一个，然后将最终的选择添加到DispatcherServlet的WebApplicationContext。如果我们使用Commons FileUpload进行文件上传，那么需要在DispatcherServlet的WebApplicationContext中添加如下bean定义：

```
<bean id="multipartResolver" ➥
class="org.springframework.Web.multipart.commons.CommonsMultipartResolver" ➥
p:maxUploadSize="1000000">
</bean>
```

25

现在，CommonsMultipartResolver（或者CosMultipartResolver）将负责分析当前multipart请求，然后将分析后的结果附着到要返回的MultipartHttpServletRequest实例（DefaultMultipartHttpServletRequest或者CosMultipartHttpServletRequest）上。当后继处理流程的Controller处理Web请求的时候，就可以使用特定的MultipartHttpServletRequest进行上传文件的获取和处理。当然，每个MultipartResolver都会有附加的属性定义以限定文件上传的行为。可以参阅Javadoc文档获得详细信息。

当MultipartResolver返回MultipartHttpServletRequest给后继处理流程，并且后继处理流程中的组件（通常是相应的Controller）也使用MultipartHttpServletRequest处理完相应的Web请求，DispatcherServlet将保证调用MultipartResolver的cleanupMultipart(..)方法，释放处理文件上传过程中所占用的系统资源。这样，整个文件上传的生命周期即告结束。

25.1.2 文件上传实践

要实现文件上传，首先按照刚才所阐述的内容，添加一个MultipartResolver的实例（或者CommonsMultipartResolver，或者CosMultipartResolver）到DispatcherServlet的WebApplicationContext中，然后再着手实际的工作。

实际上，有了MultipartResolver帮我们返回的MultipartHttpServletRequest实例，表单中的<INPUT TYPE="file" ...>元素完全可以与普通的文本域那样享受同等对待。假设我们使用扩展AbstractController的方式来处理multipart/form-data类型的如下表单提交：

```
<form:form method="post" enctype="multipart/form-data" action="fileupload.do">
```

```
    选择上传文件<input name="fileToUpload" type="file"/><br/>
    文件说明<textarea name="comment"></textarea>
    <input name="submit" type="submit" value="提交"/>
</form:form>
```

我们的Controller实现看起来与使用Servlet处理通常的Web请求没有太大差别，如代码清单25-1所示。

代码清单25-1　用于文件上传的Controller实现类代码示例

```
public class FileUploadController extends AbstractController {
    @Override
    protected ModelAndView handleRequestInternal(HttpServletRequest request, ➥
    HttpServletResponse response) throws Exception
    {
        MultipartHttpServletRequest multipartRequest = ➥
        (MultipartHttpServletRequest)request;

        MultipartFile multipartFile = multipartRequest.getFile("fileToUpload");
        String fileComment = multipartRequest.getParameter("comment");
        byte[] fileContent = multipartFile.getBytes();
        //根据情况保存到数据库或者其他地方

        ModelAndView mav = new ModelAndView();
        //添加必要的模型数据以及视图信息
        return mav;
    }
}
```

但是，在Spring MVC的Controller家族中，通常由SimpleFormController专门处理表单的提交，所以，扩展SimpleFormController才是比较合适的做法。不过在此之前，我们得为数据绑定提供一个Command对象，其定义如代码清单25-2所示。

代码清单25-2　用于文件上传的Command对象代码示例

```
public class FileUploadingCommand {
    private MultipartFile fileToUpload;
    private String comment;

    //getter和setter方法定义……

    @Override
    public String toString() {
        return new ToStringBuilder(this).append("fileToUpload", ➥
        fileToUpload).append("comment", comment).toString();
    }
}
```

在此基础上，我们定义最终的FileUploadFormController也就有了，如代码清单25-3所示。

代码清单25-3　最终用于文件上传的Controller实现类代码示例

```
public class FileUploadFormController extends SimpleFormController {
    public FileUploadFormController(){
        super.setCommandClass(FileUploadingCommand.class);
    }

    @Override
```

```
    protected void doSubmitAction(Object command) throws Exception {
        FileUploadingCommand commandBean = (FileUploadingCommand)command;
        MultipartFile multipartFile = commandBean.getFileToUpload();
        String comment = commandBean.getComment();
        FileUtils.writeByteArrayToFile(new File(".."), multipartFile.getBytes());
        // ...
    }
}
```

如果将type="file"类型的input绑定到Spring MVC框架提供的MultipartFile类型，可能会让我们的Command对象过于依赖Spring框架的API。况且，获取MultipartFile的引用之后，我们依然需要将它按照byte[]（或者数据流Stream或者字符串）的形式进行存储，所以，为了避免这些不便，我们也可以直接将Command对象中对应的字段类型声明为byte[]或者String类型。重构后的Command对象定义如代码清单25-4所示。

代码清单25-4 重构后的Command对象代码示例

```
public class FileUploadingCommand {
    private byte[] fileToUpload;
    private String comment;

    // getter和setter方法定义……

    @Override
    public String toString() {
        return new ToStringBuilder(this).append("fileToUpload", ➥
        fileToUpload).append("comment", comment).toString();
    }
}
```

为了在数据绑定过程中数据能够成功转型，我们需要为DataBinder添加相应的自定义Property-Editor实现。覆写SimpleFormController的initBinder(..)方法可以达到这个目的。工作完成后得到的最终用于文件上传的Controller实现类定义如代码清单25-5所示。

代码清单25-5 最终用于文件上传的Controller实现类代码示例

```
public class FileUploadFormController extends SimpleFormController {

    public FileUploadFormController(){
        super.setCommandClass(FileUploadingCommand.class);
    }

    @Override
    protected void doSubmitAction(Object command) throws Exception {
        FileUploadingCommand commandBean = (FileUploadingCommand)command;
        byte[] fileContent = commandBean.getFileToUpload();
        String comment = commandBean.getComment();
        FileUtils.writeByteArrayToFile(new File(".."), fileContent);
        // ...
    }

    @Override
    protected void initBinder(HttpServletRequest request, ➥
    ServletRequestDataBinder binder) throws Exception {
        super.initBinder(request, binder);
        binder.registerCustomEditor(byte[].class, new ByteArrayMultipartFileEditor());
```

25

```
    }
}
```

Spring MVC默认提供了两个自定义PropertyEditor实现类，一个就是我们刚刚使用的org.springframework.web.multipart.support.ByteArrayMultipartFileEditor，它将负责MultipartFile类型到byte[]类型的转换；另一个则是org.springframework.web.multipart.support.StringMultipartFileEditor，它负责MultipartFile类型到String类型的转换。如果使用StringMultipartFileEditor的话，Command对象中对应文件的属性需要声明为String类型，这通常对应文本文件上传的情形。

有了相应的处理文件上传的Controller之后，只需要将它们添加到DispatcherServlet的WebApplicationContext启用即可。对此，你应该已经很熟悉了。总地来说，如果不去关心细节的话，在Spring MVC中实现文件上传还是比较惬意的事情。

25.2　Handler 与 HandlerAdaptor

25.2.1　问题的起源

最初为了降低理解的难度，我们说，HandlerMapping将会通过HandlerExecutionChain返回一个Controller用于具体Web请求的处理。现在我们要进一步澄清事实：HandlerExecutionChain中所返回的用于处理Web请求的处理对象，可以不只是Controller一种类型。在Spring MVC中，任何可以用于Web请求处理的处理对象统称为Handler。Controller是Handler的一种特殊类型。HandlerMapping通过HandlerExecutionChain所返回的是一个Object类型的Handler对象，而并没限定说只能是Controller类型。所以，一般意义上讲，任何类型的Handler都可以在Spring MVC中使用，比如Struts的Action和WebWork的Action等，只要它们是用于处理Web请求的处理对象就行。

不过，对于DispatcherServlet来说，这就有点儿问题了，它如何来判断我们到底使用的是什么类型的Handler，又如何决定调用Handler对象的哪个方法来处理Web请求呢？显然，在DispatcherServlet直接硬编码if-else来枚举每一种可能的Handler类型是不具任何扩展性的。为了能够以统一的方式调用各种类型的Handler，DispatcherServlet将不同Handler的调用职责转交给了一个称为HandlerAdaptor的角色。

org.springframework.web.servlet.HandlerAdapter接口的定义如下：

```
public interface HandlerAdapter {
    boolean supports(Object handler);
    ModelAndView handle(HttpServletRequest request, HttpServletResponse response, ➥
    Object handler) throws Exception;
    long getLastModified(HttpServletRequest request, Object handler);
}
```

实际上，人如其名。哦，不对！是“接口”如其名。HandlerAdaptor将作为一个适配器，屏蔽不同Handler类型给DispatcherServlet所造成的“困扰”（所谓Adaptor Pattern应该也就是为了这种类似的目的吧）。HandlerAdapter得以成为DispatcherServlet和不同Handler的“中间人”，要归功于它的两个主要方法，即supports(..)和handle(..)。至于getLastModified(..)方法，它的主要目的只是为返回给客户端的Last-Modified这个HTTP头提供相应的时间值。如果我们不想支持该功能，直接返回-1即可。DispatcherServlet从HandlerMapping获得一个Handler之后，将询问HandlerAdaptor的supports(..)方法，以便了解当前HandlerAdaptor是否支持HandlerMapping

刚刚返回的Handler类型的调用。如果supports(..)返回true，DispatcherServlet则调用HandlerAdaptor的handle(..)方法，同时将刚才的Handler作为参数传入。方法执行后将返回ModelAndView，之后的工作就由ViewResolver接手了。现在，DispatcherServlet只需要面对HandlerAdaptor提供的统一接口，而不需要面对纷繁复杂的Handler类型。支持新的Handler类型，意味着只需要为DispatcherServlet提供新的HandlerAdaptor实现即可。DispatcherServlet借助多个HandlerAdaptor，调用当前HandlerMapping所返回的Handler来处理Web请求的逻辑，如代码清单25-6所示。

代码清单25-6 HandlerAdaptor使用场景原型代码示例（伪代码形式）

```
Object handler = <HandlerMapping对象实例>.getHandler(request).getHandler();

HandlerAdaptor handlerAdaptorToUse = null;
HandlerAdaptor[]   handlerAdaptors = ...;
for(HandlerAdaptor ha : handlerAdaptors)
{
    if(ha.supports(handler)
    {
        handlerAdaptorToUse = ha;
        break;
    }
}
if(handlerAdaptorToUse == null) ➥
throw exception for no handlerAdaptor found to use.

ModelAndView mav = handlerAdaptorToUse.handle(request,response,handler);
...
```

无论我们想在Spring MVC中使用什么类型的Handler，只要同时为DispatcherServlet提供对应该Handler的HandlerAdaptor实现就成，DispatcherServlet无须任何变动。

我想，HandlerAdaptor存在的原因我们已经搞清楚了，但还有一些问题不明，比如：

- 如果Controller只是一种特殊类型的Handler，那么Spring MVC是否还提供了其他可用的Handler类型呢？如果要提供我们自己的Handler类型又需要考虑哪些事情呢？
- 如何实现一个具体的HandlerAdaptor？Spring MVC有提供现成的实现吗？
- 如果想使用自定义的Handler，并且提供了对应的HandlerAdaptor实现，要通过什么方式告知DispatcherServlet来使用它们？

我想只有解开以上问题的答案才能帮助我们更深刻地理解Handler与HandlerAdaptor之间的关系。

25.2.2 深入了解 Handler

到目前为止，我们使用最多的Handler就是Controller。不过，如果Controller不合我们的口味的话，我们也可以使用Spring MVC提供的其他类型的Handler，甚至于自定义Handler类型。

1. 可用的Handler类型

除了Controller，Spring MVC还提供了一种与WebWork的Action功能类似的Handler实现，即org.springframework.web.servlet.mvc.throwaway.ThrowawayController。ThrowawayController接口只定义了一个方法，如下所示：

```
public interface ThrowawayController {
```

```
    ModelAndView execute() throws Exception;
}
```

ThrowawayController的奥秘倒是不在这个处理方法上。使用ThrowawayController类型的Handler的最大优势是，不需要依赖任何Servlet API，并且能够为它们定义状态，这使得ThrowawayController具有良好的可测试性。代码清单25-7给出的是一个典型的ThrowawayController实现。

代码清单25-7　ThrowawayController实现类代码示例

```
public class SuchASimpleThrowawayController implements ThrowawayController {
    private String parameter1;
    private int    parameter2;
    //其他必要的属性定义
    public ModelAndView execute() throws Exception {
        // MockService.doSthWith(parameter1,parameter2..)
        ModelAndView mav = new ModelAndView();
        //设置视图及模型数据信息
        return mav;
    }
    public String getParameter1() {
        return parameter1;
    }
    public void setParameter1(String parameter1) {
        this.parameter1 = parameter1;
    }
    public int getParameter2() {
        return parameter2;
    }
    public void setParameter2(int parameter2) {
        this.parameter2 = parameter2;
    }
    // getter和setter等方法定义
}
```

可以看到，ThrowawayController既提供处理逻辑的实现，又提供处理逻辑所需要的数据声明；框架类负责使用Servlet API提供到ThrowawayController所声明的数据的绑定；ThrowawayController实现类只需要使用绑定后的数据即可，根本不需要关心Servlet API是什么东西。

ThrowawayController不同于Controller，它拥有状态（这些状态值将由框架提供数据绑定），所以，添加到WebApplicationContext使用的ThrowawayController需要使用prototype类型的scope，而且，顾名思义ThrowawayController执行后将被丢弃，下次有新的Web请求需要同一类型的ThrowawayController处理，将获得一个新的实例。将SuchASimpleThrowawayController添加到容器后的bean定义，如下所示：

```
<bean  id="sasThrowawayController" class="..SuchASimpleThrowawayController" ➥
scope="prototype"/>
```

在Spring 2.5中，ThrowawayController已经是不推荐使用（deprecated）状态，预计将在Spring 3.0中删除，取而代之的是使用基于注解的Handler类型。如果你能够升级到Spring 2.5，那么还是选用基于注解的Handler吧！

2. 自定义Handler

可以说，自定义Handler对于Handler类型来说并没任何限制，任何我们喜欢的形式都可以。如果不喜欢Controller，也不喜欢ThrowawayController，那么可以定义自己的MyHandler，甚至不需

要强制Handler实现任何接口，仅是一个简单的POJO对象，只要能有办法知道该类就是用于Web请求处理的Handler类就行，比如用注解标注一下，然后通过反射机制就能获知那些对象是用于Web请求处理的Handler，如下所示：

```
@Handler
public class AnyType{

}
```

虽说对Handler自身没有任何限制，但是要让我们的Handler登上“历史舞台”发挥它的作用，却需要有能够给予帮助的“左膀右臂”，为我们的Handler提供必要的HandlerMapping和HandlerAdaptor这才是真正让Handler自身没有任何限制的原因所在。

HandlerMapping负责查找相应的Handler以处理Web请求。要想使用Handler，首先需要提供一个能够识别该Handler的HandlerMapping实现。比如，无论是BeanNameUrlHandlerMapping还是SimpleUrlHandlerMapping，它们都可以获取并返回Controller类型或者ThrowawayController类型的Handler。如果可以通过BeanNameUrlHandlerMapping或者SimpleUrlHandlerMapping告知DispatcherServlet我们的Handler存在的话，那还好。否则，我们就不得不提供一个能够识别我们自己Handler类型的HandlerMapping。Spring 2.5就提供了特定的DefaultAnnotation-HandlerMapping，处理新提供的基于注解的Handler的查找。

现在HandlerMapping返回了我们自定义的Handler，但DispatcherServlet本身显然是不管我们的Handler到底是何方人物的。为了让我们的Handler得以被DispatcherServlet所“青睐”，我们不得不提供一个HandlerAdaptor。实际上，并非只有我们的自定义Handler要“受此礼遇”，所有Spring MVC框架内的Handler都提供有相对应的HandlerAdaptor实现，如下所述。 25

- Controller 将 org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter作为其HandlerAdaptor。
- org.springframework.web.servlet.mvc.throwaway.ThrowawayControllerHandlerAdapter对应ThrowawayController的HandlerAdaptor实现类。
- Spring 2.5新添加的基于注解的Handler由org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter作为其HandlerAdaptor。

这是一个“传统”，我们也得遵守。不过话又说回来了，保持框架设计的统一性确实非常重要。

说了这么多，你对HandlerAdaptor可能还是没有一个感性的认识。别急，马上为你奉上。

25.2.3 近看 HandlerAdaptor 的奥秘

实际上，为具体的Handler类型提供一个HandlerAdaptor实现类非常简单。主要工作只是调用这个HandlerAdaptor“认识”的Handler的Web请求处理方法，然后将处理结果转换为DispatcherServlet统一使用的ModelAndView就行。

我们不妨就以Controller的HandlerAdaptor为例，来看一下HandlerAdaptor实现到底长什么样子。代码清单25-8给出的就是Controller对应的HandlerAdaptor实现代码。

代码清单25-8 SimpleControllerHandlerAdaptor代码摘录

```
public class SimpleControllerHandlerAdapter implements HandlerAdapter {
    public boolean supports(Object handler) {
        return (handler instanceof Controller);
```

```
    }

    public ModelAndView handle(HttpServletRequest request, HttpServletResponse response, ➥
    Object handler) throws Exception {
        return ((Controller) handler).handleRequest(request, response);
    }

    public long getLastModified(HttpServletRequest request, Object handler) {
        if (handler instanceof LastModified) {
            return ((LastModified) handler).getLastModified(request);
        }
        return -1L;
    }
}
```

SimpleControllerHandlerAdapter是Controller对应的HandlerAdaptor实现。supports(..)方法决定了它只认识Controller一个人，所以，在handle(..)方法中，直接将Object类型的Handler强制转型为Controller，然后调用其handleRequest(..)即可。因为Controller的handleRequest(..)方法可以返回已经组装好的ModelAndView，所以，就直接返回了。

有了SimpleControllerHandlerAdapter，我们是否可以给出更多的实现了呢？比如，假设我想在Spring MVC中使用Struts 1.x的Action作为Handler，那么是否可以提供一个类似代码清单25-9所示的StrutsActionHandlerAdaptor呢？

代码清单25-9　处理Struts Action的HandlerAdaptor原型实现代码示例

```
public class StrutsActionHandlerAdaptor implements HandlerAdapter {
    public boolean supports(Object handler) {
        return (handler instanceof Action);
    }
    public ModelAndView handle(HttpServletRequest request, ➥
    HttpServletResponse response, Object handler) throws Exception {
        Action action = (Action)handler;

        ActionMapping actionMapping = getActionMapping();
        ActionForm actionForm = createAndBindToActinForm(request,response);
        ActionForward forward = action.execute(actionMapping, actionForm, request, response);
        View view = constructViewAccordingToActionForward(forward);
        return new ModelAndView(view);
    }

    public long getLastModified(HttpServletRequest req, Object handler) {
        return -1;
    }

    private ActionForm createAndBindToActinForm(HttpServletRequest request, ➥
    HttpServletResponse response) {
        // ...
    }
    private ActionMapping getActionMapping() {
        // ...
    }
    private View constructViewAccordingToActionForward(ActionForward forward)
    {
```

```
        // ...
    }
}
```

显然，只要能够给出对应的HandlerAdaptor实现，任何类型的Handler都可以纳入Spring MVC使用。

25.2.4 告知 Handler 与 HandlerAdaptor 的存在

我们已经有了要添加到Spring MVC框架的新的Handler类型，也给出了针对该Handler类型的HandlerAdaptor实现类，现在是让DispatcherServlet接纳它们的时候了！

首先，如果现有的HandlerMapping不足以“感知”到我们的Handler类型的话，那么我们需要提供一个能够“感知”我们Handler的HandlerMapping实现类，并将其注册到DispatcherServlet的WebApplicationContext中。如果现有的HandlerMapping实现可以“感知”到我们的Handler，那么将可以省去实现自定义HandlerMapping的工作，但依然需要将使用的HandlerMapping添加到DispatcherServlet的WebApplicationContext中，除非我们使用默认的BeanNameUrlHandlerMapping。大部分情况下，只要我们提供的Handler在容器中的引用，能够明确指定给BeanNameUrlHandlerMapping或者SimpleUrlHandlerMapping等现有HandlerMapping实现类，都不需要自定义HandlerMapping实现。当然，在稍后详细介绍Spring 2.5新添加的基于注解的Handler实现的时候，我们将看到第一个为特定Handler提供的HandlerMapping实现。

其次，有了可以返回我们自定义的Handler的HandlerMapping之后，我们要为DispatcherServlet提供能够调用该类型Handler的HandlerAdaptor实现。这同样是通过将HandlerAdaptor实现类添加到WebApplicationContext完成的，类似如下的bean定义：

```
<bean id="anyName" class="..MyHandlerAdaptor">
    <!--必要的属性设置-->
</bean>
```

可以向DispatcherServlet的WebApplicationContext中添加多个HandlerAdaptor。DispatcherServlet将根据类型自动检测容器内可用的HandlerAdaptor实例。如果无法找到可用的HandlerAdaptor，DispatcherServlet将启用后备的几个默认使用的HandlerAdaptor实现，它们是：

- org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter[①]；
- org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter；
- org.springframework.web.servlet.mvc.throwaway.ThrowawayControllerHandlerAdapter；
- org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter。

也就是说，如果使用的Handler是现有Handler类型，那么无须在DispatcherServlet的WebApplicationContext中做任何配置，默认的HandlerAdaptor已经足够了。不过，如果需要添加这些HandlerAdaptor类型之外的HandlerAdaptor实现，并且我们依然希望同时使用这些默认HandlerAdaptor所支持的Handler的话，那就需要在添加我们的自定义HandlerAdaptor的基础上，同时添加以上几种默认的HandlerAdaptor实现，如代码清单25-10所示。

① 我们将在Spring Remoting一章近距离接触该类型的handler。

代码清单25-10　添加多个HandlerAdaptor的配置代码示例

```
<bean id="anyName" class="..MyHandlerAdaptor">
    <!--必要的属性设置-->
</bean>

<bean id=".." class="org.springframework.Web.servlet.mvc.HttpRequestHandlerAdapter">
    <!--必要的属性设置-->
</bean>

<bean id=".." class="org.springframework.Web.servlet.mvc.SimpleControllerHandlerAdapter">
    <!--必要的属性设置-->
</bean>
<bean id=".." ➥
class="org.springframework.Web.servlet.mvc.throwaway.ThrowawayControllerHandlerAdapter">
    <!--必要的属性设置-->
</bean>
<bean id=".." ➥
class="org.springframework.Web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter">
    <!--必要的属性设置-->
</bean>
```

最后，不要忘了把每个Handler都添加到DispatcherServlet的WebApplicationContext中。当然，如果你的HandlerMapping不需要容器的支持也可以返回一个可用的Handler实例，那么这一步可以免掉。

不管怎么说，有了以上的工作，现在使用什么类型的Handler类来处理Web请求就可以随心所欲了。

25.3　框架内处理流程拦截与HandlerInterceptor

前面已经讲述了，HandlerMapping返回的用于处理具体Web请求的Handler对象，是通过一个HandlerExecutionChain对象进行封装的（这在HandlerMapping的接口定义上可以看出来），我们却一直没有对这个HandlerExecutionChain做进一步的解释，现在是彻底揭开这个谜团的时候了。

说白了，HandlerExecutionChain就是一个数据载体，它包含了两方面的数据，一个就是用于处理Web请求的Handler，另一个则是一组随同Handler一起返回的HandlerInterceptor。这组HandlerInterceptor可以在Handler的执行前后对处理流程进行拦截操作。

HandlerInterceptor定义了如下三个拦截方法：

```
public interface HandlerInterceptor {
    boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object ➥
    handler) throws Exception;
    void postHandle(HttpServletRequest request, HttpServletResponse response, Object ➥
    handler, ModelAndView modelAndView) throws Exception;
    void afterCompletion(HttpServletRequest request, HttpServletResponse response, ➥
    Object handler, Exception ex) throws Exception;
}
```

下面是对这三个拦截方法的简单说明。

- **boolean preHandle(..)**。该拦截方法将在相应的HandlerAdaptor调用具体的Handler处理Web请求之前执行。如果想在此之前阻断后继处理流程，preHandle(..)方法将是最合适也是我们唯一的选择。preHandle(..)通过boolean返回值表明是否继续执行后继处理流程，如下所述。

- true表明允许后继处理流程继续执行。如果当前HandlerInterceptor位于所在HandlerInterceptor链之前或者中间位置，那么后继HandlerInterceptor的preHandle(..)将继续执行。如果HandlerInterceptor是所在HandlerInterceptor链的最后一个，那么处理Web请求的Handler将允许执行。
- false表明preHandle(..)方法不允许后继流程的继续执行，包括HandlerInterceptor链中的其他HandlerInterceptor以及其后的Handler。在这种情况下，通常认为preHandle(..)方法内部已经自行处理掉了当前的Web请求。当然，通过抛出相应的异常的方式，也可以达到与返回false同样的阻断效果。

一般来说，preHandle(..)将是我们使用最多的拦截方法。我们也可以在这里进行一些必要条件检查，如果没能通过检查，通过preHandle(..)可以阻断后继处理流程的执行。

- **void postHandle(..)**。该拦截方法的执行时机为HandlerAdaptor调用具体的Handler处理完Web请求之后，并且在视图的解析和渲染之前。通过该方法我们可以获取Handler执行后的结果，即ModelAndView。我们可以在原处理结果的基础上对其进行进一步的后处理，比如添加新的统一的模型数据，或者对ModelAndView中的数据进行变更等。postHandle(..)返回类型为void，不可以阻断后继处理流程。
- **void afterCompletion(..)**。在框架内整个处理流程结束之后，或者说视图都渲染完了的时候，不管是否发生异常，afterCompletion(..)拦截方法将被执行。如果处理是异常结束的话，我们可以在该方法中获得异常（Exception）的引用并对其进行统一处理。另外，如果Web请求处理过程中有相应资源需要清理的话，也可以在这里完成。不用说也知道，afterCompletion(..)的返回值为void，并且到它执行的时候，处理流程已经是尾声了，根本没有阻断执行流程的必要。

我想，对于HandlerInterceptor三个拦截方法的说明已经很明了了。如果要对HandlerInterceptor的拦截位置有一个更加感性的认识的话，不妨回头看一下我们出发前的地图（图25-2）。

25

25.3.1 可用的 HandlerInterceptor 实现

做任何事情之前我们都会先去找一下有没有现成的“锤子”。对于HandlerInterceptor来说，情况同样如此。在实现自定义的HandlerInterceptor之前，我们先看一下Spring MVC都准备了哪些现成的HandlerInterceptor实现。

实际上，通过查看HandlerInterceptor的继承层次，我们可以发现四五个HandlerInterceptor实现类：UserRoleAuthorizationInterceptor、WebContentInterceptor、LocaleChangeInterceptor以及ThemeChangeInterceptor等。不过，鉴于在稍后介绍LocalResolver和ThemeResolver的时候会再次接触LocaleChangeInterceptor和ThemeChangeInterceptor，我们先将它们放置一边，重点看一下UserRoleAuthorization Interceptor和WebContentInterceptor这两个可用的HandlerInterceptor实现类。

org.springframework.web.servlet.handler.UserRoleAuthorizationInterceptor。UserRoleAuthorizationInterceptor允许我们通过HttpServletRequest的isUserInRole(..)方法，使用指定的一组用户角色（UserRoles）对当前请求进行验证。如果验证通不过，UserRoleAuthorizationInterceptor将默认返回HTTP的403状态码，即forbidden。我们可以通过覆写handleNotAuthorized(..)方法改变这种默认行为，比如将请求导向一个信息页面。UserRoleAuthorizationInterceptor的使用极其简单，只需要指定验证用的一组用户角色（UserRoles）即可，如下所示：

```
<bean id="userRolesAuthHandlerInterceptor" ➥
class="org.springframework.Web.servlet.handler.UserRoleAuthorizationInterceptor">
    <property name="authorizedRoles">
        <list>
            <value>Admin</value>
            ...
        </list>
    </property>
</bean>
```

UserRoleAuthorizationInterceptor将循环遍历这组指定的用户角色（UserRoles）对当前请求进行验证。

org.springframework.web.servlet.mvc.WebContentInterceptor。WebContentInterceptor对处理流程的拦截主要做如下几件事情。

- **检查请求方法类型是否在支持方法之列。** 如果当前请求的方法类型超出我们通过setSupportedMethods(..)方法指定的范围，那么WebContentInterceptor将抛出HttpRequestMethodNotSupportedException从而阻断后继处理流程。这通常用于进一步限定请求的方法类型，比如，我们可以通过setSupportedMethods(..)方法设置supportedMethods只为POST一种，不支持GET或者其他请求方法类型。
- **检查必要的Session实例。** 如果我们设置requireSession属性为true，同时又发现当前请求不能返回一个已经存在的Session实例，WebContentInterceptor将抛出HttpSessionRequiredException阻断后继处理流程。
- **检查缓存时间并通过设置相应HTTP头（Header）的方式控制缓存行为。** WebContentInterceptor允许我们通过setCacheSeconds(..)方法设置请求内容的缓存时间。它将通过设置用于缓存管理的HTTP头（HTTP Header）的形式，对请求内容对应的缓存行为进行管理。我们可以通过useCacheControlHeader或者useExpiresHeader属性，进一步明确是使用的HTTP 1.1的Cache-Control指令还是HTTP 1.0的Expires指令。

通常，WebContentInterceptor使用方式如下所示：

```
<bean id="WebContentInterceptor" ➥
class="org.springframework.Web.servlet.mvc.WebContentInterceptor" ➥
p:cacheSeconds="30" p:supportedMethod="POST">
</bean>
```

除此之外，我们还可以通过setCacheMappings(..)方法，进一步明确指定不同请求与其缓存时间之间的细粒度的映射关系。

> **注意**　UserRoleAuthorizationInterceptor和WebContentInterceptor都是只在preHandle(..)拦截方法中实现了相应的拦截逻辑。我想，你应该已经从它们能够“阻断后继处理流程”的功能上看出这一点。

25.3.2　自定义实现 HandlerInterceptor

Spring为我们提供了现成的HandlerInterceptor固然不错，但这并不足以满足广大群众的各种需求。单就HandlerInterceptor作为一个扩展点而存在的意义来讲，如果拦截Web请求处理逻辑的需求就那么几种的话，完全没有必要设置这么一个角色。而实际上，我们所要面对的系统和场景却是繁杂多变的，所以，大部分时间，我们不得不根据应用的需求提供我们的自定义HandlerInterceptor

实现类。

在给出我们的自定义HandlerInterceptor之前，按照惯例，我们还得先把该HandlerInterceptor所要处理的业务场景说清楚。FX系统宣称全天24小时都可交易，但实际上，每天都会有少量的时间用于做rollover操作，以便对系统状态进行调整。而这段时间是不能为顾客提供交易服务的。所以，我们要对顾客的交易行为进行限制，当市场处于关闭状态的时候，将不予处理用户的请求。为了达到该目的，可以从程序逻辑上，或者系统管理上，进行多方面考虑。但既然我们现在处于HandlerInterceptor的“领地”，就把这块儿工作交给HandlerInterceptor做如何？

我们的自定义HandlerInterceptor实现类有一个工作，如果发现现在市场是处于关闭状态，则将当前Web请求转向一个表明系统暂停交易画面，否则允许请求的处理继续进行，如代码清单25-11所示的原型实现。

代码清单25-11 自定义HandlerInterceptor实现类代码示例

```
public class MarketAccessInterceptor extends HandlerInterceptorAdapter {
    private static final Log logger = LogFactory.getLog(MarketAccessInterceptor.class);

    private String marketClosedPageUrl;
    private IMarketStatusMonitor marketStatusMonitor;

    @Override
    public boolean preHandle(HttpServletRequest request,
        HttpServletResponse response, Object handler) throws Exception {
        if(getMarketStatusMonitor().isMarketClosed())
        {
            InternalResourceView view = new InternalResourceView();
            view.setUrl(getMarketClosedPageUrl());
            view.render(null, request, response);
            if(logger.isInfoEnabled())
            {
                logger.info("since market has been close, " + ➥
                "we will redirect the user request to market-close information page.");
            }
            return false;
        }
        return true;
    }
    // getter和setter方法定义……
}
```

25

IMarketStatusMonitor提供了检查市场状态所需要的服务，至于是让它通过检查数据库状态还是检查其他数据源来完成这项使命，那就不是我们的MarketAccessInterceptor所要关心的啦！

> **提示** 我们的MarketAccessInterceptor并没有直接实现HandlerInterceptor接口，而是继承自HandlerInterceptorAdapter类。不知道你是否已经明了其中缘由？
> 实际上，如果你足够细心的话，就会发现这是Java平台API设计上的一个“光荣传统”。对于那些需要经常被扩展，而又包含多个方法需要实现的接口声明，通常情况下，使用者并非每次都希望提供所有接口方法的实现逻辑。为了避免在实现这些接口的时候，每次都要去实现接口定义中的所有方法，对应API的设计者通常都会提供一个XXXAdaptor类专门用于子类化的需要，避免实现所有接口方法的烦琐。
> 比如，在AWT/Swing库中的各种事件监听器（EventListener）接口定义，通常都会定义多个

事件回调方法。java.awt.event.MouseListener接口就定义了如下5个事件回调方法。

- void mouseClicked(MouseEvent e)。
- void mousePressed(MouseEvent e)。
- void mouseReleased(MouseEvent e)。
- void mouseEntered(MouseEvent e)。
- void mouseExited(MouseEvent e)。

但通常情况下，我们可能只希望处理鼠标点击等一两个动作，那么每次直接实现MouseListener接口则比较烦琐。MouseAdapter将帮助我们减轻子类化时候的“压力”。我们直接继承MouseAdapter，然后覆写需要实现的那个方法即可。

我们还可以在其他地方遇到这种提供XXXAdaptor的API设计方式。如果某一天我们也遇到类似的情况的话，说不定也可以借鉴一下哦。只不过，不要将它与Adaptor模式（Adaptor Pattern）里的那个Adaptor相混淆。

有了MarketAccessInterceptor自定义实现类，我们就可以将其添加到WebApplicationContext使用了，如下所示：

```
<bean id="marketAccessInterceptor" ➥
class="cn.spring21.simplefx.interceptor.MarketAccessInterceptor">
    <property name="marketClosedPageUrl" ➥
    value="/WEB-INF/jsp/marketCloseNotification.jsp"/>
    <property name="marketStatusMonitor" ref="marketStatusMonitor"/>
</bean>
```

marketClosedPageUrl属性指定市场关闭后请求将被转向的信息页面路径。因为我们在MarketAccessInterceptor内部通过InternalResourceView来渲染视图，所以指定的路径指向.jsp形式的视图模板。不过，如果想要MarketAccessInterceptor拥有更好的扩展性，改为直接注入相应的视图也无妨。

25.3.3 HandlerInterceptor 寻根

我们已经知道了HandlerInterceptor的作用，也知道了Spring MVC都提供了哪些常用的HandlerInterceptor实现，甚至，也了解了如何自定义一个HandlerInterceptor，却还不知道到底应该如何将HandlerInterceptor纳入应用程序才能使其工作。

HandlerInterceptor和Handler实际上“本是同根生”。如果我们从HandlerInterceptor所处的位置溯源而上（按照HandlerInterceptor→HandlerExecutionChain→HandlerMapping的顺序），则会发现HandlerMapping是其最终的发源地，AbstractHandlerMapping作为几乎所有HandlerMapping实现类的父类，提供了setInterceptors(..)方法以接受一组指定的HandlerInterceptor实例。所以，要使我们的HandlerInterceptor发挥作用，只要将它添加到相应的HandlerMapping即可，如下所示：

```
<bean id="handlerMapping" ➥
class="org.springframework.Web.servlet.handler.BeanNameUrlHandlerMapping">
    <property name="interceptors">
        <list>
            <ref bean="userRolesAuthHandlerInterceptor"/>
            <ref bean="WebContentInterceptor"/>
```

```
            <ref bean="marketAccessInterceptor"/>
        </list>
    </property>
</bean>
```

这些指定的HandlerInterceptor将随同处理具体Web请求的Handler一起返回（以Handler-ExecutionChain的形式），并对Web请求的处理流程进行拦截。

在讲解HandlerMapping的时候提到过，我们可以在同一Spring MVC应用程序中指定多个拥有不同优先级的HandlerMapping，每一HandlerMapping管理自己所负责的一组Web请求处理的映射关系。在HandlerInterceptor结合HandlerMapping的这一特性（即Chaining Of HandlerMapping）之后，就可以细化Web请求处理流程的拦截范围。我们可以只赋予某一HandlerMapping相应的Handler-Interceptor实例，这样，该HandlerInterceptor就可以只拦截这一HandlerMapping所管理的那一组处理Web请求的Handler。相对于只有一个HandlerMapping的情况，这样的处理方式让HandlerInterceptor的使用更加地灵活。

25.3.4 HandlerInterceptor 之外的选择

在Spring MVC中，并非只有HandlerInterceptor才能对Web请求的处理流程进行拦截并做相应的处理。既然Spring MVC同样基于Servlet API构建，那么，Servlet提供的规范设施自然也可以在使用Spring MVC的Web应用程序中使用。所以，能够提供拦截能力的Servlet标准组件Filter，就成为了使用HanlderInterceptor之外的选择。

HandlerInterceptor和Servlet Filter的共同特点是都可以用于Web请求处理流程的拦截。但在某些方面二者也存在一些差别，如图25-3所示。

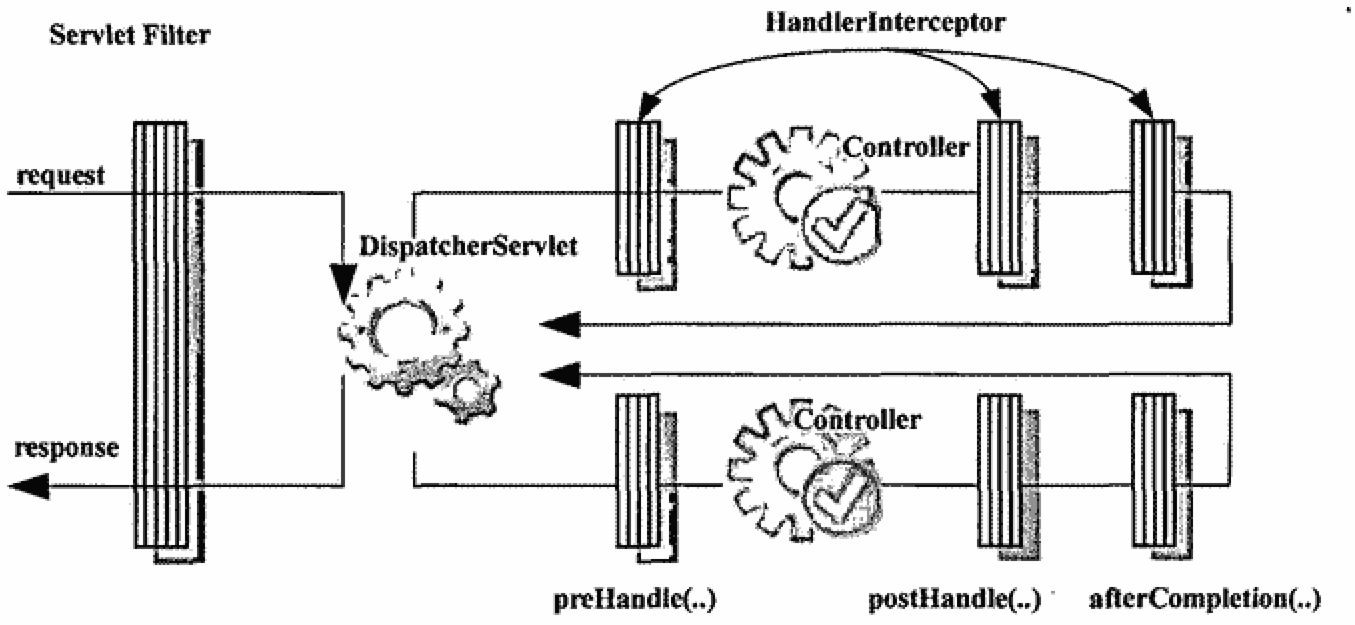

图25-3　Servlet Filter和HandlerInterceptor关系示意图

HandlerInterceptor位于DispatcherServlet之后，指定给HandlerMapping的它，可以对HandlerMapping所管理的多组映射处理关系进行拦截。最主要的是，HandlerInterceptor拥有更细粒度的拦截点。我们可以在Handler执行之前，Handler执行之后，以及整个DispatcherServlet内部处理流程完成时点插入必要的拦截逻辑。通过结合HandlerMapping的Chaining特性，我们可以对不同HandlerMapping管理的多组不同的Handler，应用不同的HandlerInterceptor进行处理流程

的拦截处理，总之，HandlerInterceptor带给我们的是一种具有高灵活性的细粒度的请求处理流程拦截方案。

与HandlerInterceptor不同，Filter提供的拦截方式更像是一种对Web应用程序的“宏观调控”。作为Servlet规范的标准组件，Filter通常被映射到Java Web应用程序中的某个servlet，或者一组符合某种URL匹配模式的访问资源上。所以，从Spring MVC应用的结构上看，Filter位于DispatcherServlet之前。如果把Filter和HandlerInterceptor看作同一类型的拦截器，Filter将比HandlerInterceptor拥有更高的执行优先级。不过，二者提供的拦截功能所加诸于上的目标对象却完全是不同级别：Filter序列在servlet层面对DispatcherServlet进行拦截，而HandlerInterceptor则位于DispatcherServlet内部，对Handler的执行进行拦截。Filter的应用位置注定了它不能够提供细粒度的拦截时点，所以，通常情况下，使用Filter对于Web应用程序中的一些普遍关注点进行统一处理是比较适合的，一旦需要细化处理流程的拦截逻辑，可以再转而求助于Handlerinterceptor。

Filter是servlet标准组件，需要在web.xml中配置，这就意味着，其生命周期管理更多是由Web容器进行管理的。如果我们的Filter在实现期间需要某些服务的支持，尤其是当前Spring MVC应用的WebApplicationContext中的某些服务的支持，我们不得不采用某种过度耦合的绑定机制或者查找方式来获取这些服务的支持。为了能够让Filter的实现更加无拘无束，尽情享用依赖注入所带来的乐趣，Spring MVC引入了org.springframework.web.filter.DelegatingFilterProxy以改变Filter的现状。

顾名思义，DelegatingFilterProxy的作用是作为一个Filter的Proxy对象，当真正需要执行拦截操作的时候，它将把具体的工作委派给它所对应的一个Filter委派对象。在物理结构上，DelegatingFilterProxy位于web.xml中承担Filter的原始使命，而它所委派的那个Filter对象，也就是做实际工作的那个家伙，却可以置身于WebApplicationContext中，充分享受Spring的IoC容器所提供的各项服务。图25-4演示了DelegatingFilterProxy与其对应的Filter实例之间的存在关系。

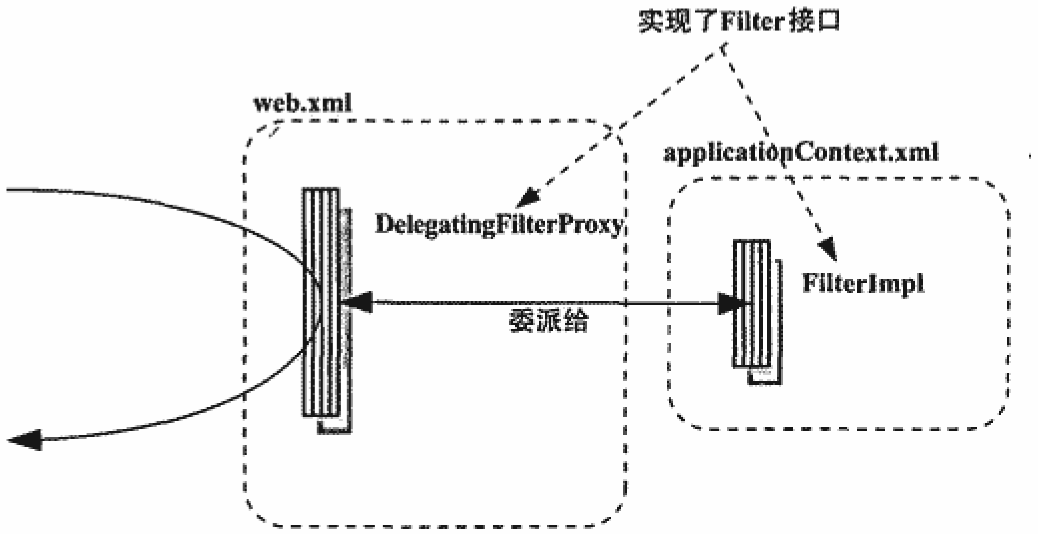

图25-4　DelegatingFilterProxy与其Filter委派对象之间的关系

在DelegatingFilterProxy和其Filter委派对象的关系中，基本上所有“脏活儿”（dirty work）都是由DelegatingFilterProxy来做的，包括从绑定到ServletContext的WebApplicationContext中获取其将使用的Filter委派对象，然后读取原始的Filter配置信息，并设置给委派对象使用等。只有在这些准备工作都完成之后，我们才能使用真正依赖的那个Filter对象。这是不是让我们感觉又回到了IoC之前的时代了呢？我们的最终目的只是想要一个直接可用的Filter实例而已。不过，Delegating-

FilterProxy做了这些事情，倒是能够解脱实际使用的那个Filter实现类，也算值得吧！有了DelegatingFilterProxy的付出，我们自己的Filter实现过得就舒服多了。代码清单25-12给出了一个享受IoC服务的Filter实现类。

代码清单25-12　配合DelegatingFilterProxy使用的Filter实现类代码示例

```
public class AnyFilter implements Filter {
    private IAuthorityService authorityService;
    private IAnySerivce       anyServiceYouWant;

    public void init(FilterConfig config) throws ServletException {
        //如果需要的话，做必要的初始化工作
    }

    public void doFilter(ServletRequest request, ServletResponse response, ➥
    FilterChain filterChain) throws IOException, ServletException {
        //使用注入的服务实现相应的逻辑
        filterChain.doFilter(request, response);
    }

    public void destroy() {
        //可能必要的对象销毁逻辑
    }

    public IAuthorityService getAuthorityService() {
        return authorityService;
    }

    public void setAuthorityService(IAuthorityService authorityService) {
        this.authorityService = authorityService;
    }

    public IAnySerivce getAnyServiceYouWant() {
        return anyServiceYouWant;
    }

    public void setAnyServiceYouWant(IAnySerivce anyServiceYouWant) {
        this.anyServiceYouWant = anyServiceYouWant;
    }
}
```

只要将我们自己的Filter实现添加到WebApplicationContext管理，该Filter实现类就可获得所需要的依赖注入，如下所示：

```
<bean id="anyCustomFilter" class="..AnyFilter">
    <property name="authorityService" ref=".."/>
    <property name="anyServiceYouWant" ref=".."/>
</bean>
```

不过，要想让这个原来本该在web.xml中配置生效，而现在却在WebApplicationContext中的Filter发挥作用，我们需要让它与DelegatingFilterProxy发生关系才行。而让WebApplicationContext中的Filter实例成为DelegatingFilterProxy的代理对象的默认关系标志，就是容器中Filter实例的beanName，只需要保证其beanName与DelegatingFilterProxy所对应的<filter-name>的值相同，就能确定二者之间的委派关系。代码清单25-13给出的是对应的DelegatingFilterProxy在web.xml中的配置内容。

代码清单25-13 对应的DelegatingFilterProxy的配置代码示例

```
<filter>
    <filter-name>anyCustomFilter</filter-name>
    <filter-class>
        org.springframework.Web.filter.DelegatingFilterProxy
    </filter-class>
</filter>

<filter-mapping>
    <filter-name>anyCustomFilter</filter-name>
    <url-pattern>..</url-pattern>
    <!-- 或者 <servlet-name>..</servlet-name>-->
</filter-mapping>
```

DelegatingFilterProxy将根据<filter-name>的值到WebApplicationContext中抓取对应的Filter实例作为其委派对象来使用，在这里也就是“anyCustomerFilter”所对应的Filter实例。

在使用DelegatingFilterProxy的场景中，具体Filter实例的生命周期默认是由WebApplicationContext容器进行管理的。也就是说，Filter的标准生命周期管理方法init(..)和destroy()，对于WebApplicationContext容器来说没有特殊的意义。DelegatingFilterProxy默认不负责调用具体Filter实例的这两个方法。如果我们希望改变这一场景中默认的生命周期管理方式，即希望由默认的WebApplicationContext容器管理转向原始的Web容器管理，那么，我们可以通过将DelegatingFilterProxy的targetFilterLifecycle属性值从默认的false设置为true以完成这一转变，如下所示：

```
<filter>
    <filter-name>anyCustomFilter</filter-name>
    <filter-class>
        org.springframework.Web.filter.DelegatingFilterProxy
    </filter-class>
    <init-param>
        <param-name>targetFilterLifecycle</param-name>
        <param-value>true</param-value>
    </init-param>
</filter>
```

DelegatingFilterProxy从WebApplicationContext中获取到Filter委派对象实例之后，将负责调用该委派对象的init(..)初始化方法，销毁的操作也是同样道理。

注意 除DelegatingFilterProxy之外，Spring MVC还在org.springframework.web.filter包中提供了多个Filter实现类，我们可以在实际开发中根据情况选用它们。关于这些Filter类的详细情况，可参考Spring框架提供的Javadoc文档。

25.4 框架内的异常处理与HandlerExceptionResolver

在介绍下一位出场选手之前，我们先来简单回顾一下之前讲述的两种Handler类型的定义，即Controller和ThrowawayController，如下所示：

```
public interface Controller {
    ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse ➥
    response) throws Exception;
}
```

```
public interface ThrowawayController {
    ModelAndView execute() throws Exception;
}
```

在*Effictive Java*一书中，作者对异常处理提出了如下的一段陈述：

> 总是要单独的声明被检查的异常，并且利用Javadoc的@throws标记准确地记录下每个异常被抛出的条件。如果一个方法可能会抛出多个异常类，那么不要使用“快捷方式”，即声明它会抛出这些异常类的某个基类。作为一个极端的例子，永远不要声明一个方法“throws Exception”，或者更差的做法，“throws Throwable”。这样的声明没有为你的客户提供关于“这个方法能够抛出哪些异常”的任何指导信息，而且大大地妨碍了该方法的使用，因为它实际上掩盖了在同样的执行环境中该方法可能会抛出的任何其他异常。

对于上面的Handler处理方法定义来说，直接抛出异常的做法看起来直接违反了这段描述所倡导的异常处理最佳实践标准，而且框架开发者也承认这一点。不过，让我们换一个角度再来看这样的接口设计。

作为框架类的Handler，其应用的场景可能千差万别，而且在处理各个场景的Web请求的过程中，Handler自身或者Handler所依赖的各种业务对象所可能抛出的checked exception[①]也是不一而足。试想，如何让Handler来预知那些可能抛出的异常类型呢？就好像牛顿当年只能感慨“我能计算出天体的运行轨迹，却难以预料到……”一样，如果让Handler还去走最佳实践的那条路，显然后果也好不到哪里去。而且，应该根本就达不到最初所设想的目的。所以，框架实现者可能不得不“退而求其次”，转而throws Exception。而且，这并非尽是坏处，现在的Handler接口不会对所有可能抛出的异常类型做任何的限制。

虽然最为顶层的Handler接口定义直接throws Exception，但如果愿意，我们依然可以通过覆写Handler的子类来进一步限定处理方法可能抛出的异常类型，例如：

```
public class ExceptionRaisingController extends AbstractController {
    @Override
    protected ModelAndView handleRequestInternal(HttpServletRequest arg0, ➥
    HttpServletResponse arg1) throws TradeRateExpireException {
        throw new TradeRateExpireException("test exception");
    }
}
```

对于类似的子类实现，使用它们的客户端代码同样可以明确所要处理的具体异常情况。

当然，Handler接口定义能够如此设计和实现，其背后最强大的支持者当属即将登场的HandlerExceptionResolver。是它提供的框架内统一的异常处理方式，让throws Exception看起来更加“理直气壮”。

org.springframework.web.servlet.HandlerExceptionResolver对异常的处理范围仅限于Handler查找以及Handler执行期间，也就是图25-2中矩形所圈定的范围。HandlerExceptionResolver和Handler的关系最不一般，它们就好像双子座两兄弟一样，如果Handler执行过程中没有任何异常，将以ModelAndView的形式返回后继流程要用的视图和模型数据信息，而一旦出现异常情况，HandlerExceptionResolver将接手异常情况的处理，处理完成后，将同样以ModelAndView的形式返回后继处理流程要使用的视图和模型数据信息。只不过，HandlerExcep-

① 系统中面向业务的异常类型通常都设计为checked exception形式。

tionResolver所返回的ModelAndView中所包含的信息是错误信息页面和相关异常的信息。试想一下，不是兄弟，HandlerExceptionResolver为什么要专门替Handler收拾各种烂摊子呢？

我们已经了解了Handler，下面来详细见识一下HandlerExceptionResolver吧！其定义如下：

```
public interface HandlerExceptionResolver {
    ModelAndView resolveException(HttpServletRequest request, ➥
    HttpServletResponse response, Object handler, Exception ex);
}
```

HandlerExceptionResolver仅定义了一个resolveException(..)方法，负责处理与相关Handler所关联的某种异常（如参数Exception ex所示），并异常处理完成后，将处理结果以ModelAndView的形式返回。这当然包括将要跳转到的错误信息页面，以及该页面所要显示的必要信息。至于HandlerExceptionResolver所返回的ModelAndView的后继处理，与Handler处理后返回的ModelAndView应该说是“殊途同归”了。如果没记错的话，下面DispatcherServlet将寻求ViewResolver和View对这些返回的信息进行处理。

从HandlerExceptionResolver的“家谱”上看，它在Spring MVC框架中有点儿“人丁不旺”的意思。因为框架只提供了org.springframework.web.servlet.handler.SimpleMappingExceptionResolver这一个可用的实现类。不过，通常情况下，SimpleMappingExceptionResolver已经足够完成主要使命了。

SimpleMappingExceptionResolver使用Properties来管理具体异常类型与所要转向的错误页面之间的映射对应关系。大部分情况下，通过Properties类型的exceptionMappings属性指定具体的映射关系，将是使用SimpleMappingExceptionResolver进行异常处理所要做的主要的工作。代码清单25-14给出的正是SimpleMappingExceptionResolver的使用配置示例。

代码清单25-14　SimpleMappingExceptionResolver的使用配置代码示例

```
<bean name="handlerExResolver" ➥
class="org.springframework.Web.servlet.handler.SimpleMappingExceptionResolver">
    <property name="defaultErrorView" value="defaultErrorViewName"/>
    <property name="exceptionMappings">
        <props>
            <prop key="TradeRateExpireException">tradeRateExpireErrorPage</prop>
            ...
            <prop key="java.lang.Exception">..</prop>
        </props>
    </property>
</bean>
```

SimpleMappingExceptionResolver内部将遍历exceptionMappings的所有元素，寻找其中与当前抛出的异常类型“最接近”的映射项，并将其对应的值作为错误信息页面的逻辑视图名，然后封装到ModelAndView中返回以供后继处理流程使用。

> **小心**　SimpleMappingExceptionResolver在将当前抛出的异常类型与exceptionMappings中相应映射进行匹配的时候，采用的不是类型匹配，而是根据类名进行匹配，更进一步说，是使用类名字符串的局部匹配原则。打个比方吧！假设我们有如下映射关系：

```
Exception <-> page1
TradeRateExpireException <-> page2
exceptions.TradeRateExpireException <-> page3
```

如果当前抛出的异常类型是a.b.c.exceptions.TradeRateExpireException，那么最终匹配结果可能不是我们想要的page2，而是page1。为什么会造成这种情况呢？SimpleMappingExceptionResolver使用String的indexOf(..)方法来判断当前抛出异常类型的类名a.b.c.exceptions.TradeRateExpireException中是否存在exceptionMappings所指定的各项映射中的键值。存在的话，则首先认为存在的映射项为匹配结果，并且同时限定查找的depth的上下限。如果余下的其他映射项的匹配不足以进一步缩小depth的上下限，那么最初的匹配结果将作为最终结果。在我们的映射中，对Exception的匹配已经将depth的上下限限定在了[0,0]，所以，即使下面有更加合适的匹配结果，也不能够进一步缩小该depth上下限范围，那么Exception对应的page1就作为结果返回。而实际上，我们想要的是TradeRateExpireException，甚至于最差也应该是exceptions.TradeRateExpire-Exception。显然SimpleMappingExceptionResolver这种处理方式并不合适。不过，如果你要使用SimpleMappingExceptionResolver的话，则需要了解该实现类在这部分逻辑实现上的"陷阱"，以免预料之外的情况发生了，却找不到原因①。

为了避免SimpleMappingExceptionResolver的这种indexOf(..)匹配方式可能造成的"误解"，建议在指定映射中的异常名称的时候，全部使用全限定类名（Full Qualified Class Name）。这样，就会进一步强制SimpleMappingExceptionResolver去做类名的完全匹配，从而最大限度地避免以上问题的发生。

25

除了通过exceptionMappings属性定制SimpleMappingExceptionResolver的异常处理行为，SimpleMappingExceptionResolver还定义了其他几个属性，以进一步定制其功能，主要为如下几个。

- **defaultErrorView**。用于指定一个默认的错误信息页面对应的逻辑视图名。当无法通过exceptionMappings查找到可用的视图名的时候，defaultErrorView所指定的默认视图名将被返回。
- **defaultStatusCode**。可以指定异常情况下默认返回给客户端的HTTP状态码。比如，我们可以通过defaultStatusCode设定默认的HTTP状态码为500（SC_INTERNAL_ SERVER_ERROR）或者404（SC_NOT_FOUND）。当然，该属性是可选的。
- **exceptionAttribute**。如果我们想要在错误信息页面中对抛出的异常进行访问，那么可以设定exceptionAttribute的值，然后以设定后的值作为键，在错误信息页面中获取抛出的异常实例。当然，即使不对其进行设定，我们依然可以在错误信息页面中使用exception-Attribute的默认值"exception"获取相应的异常实例。另外，如果不想将异常公开给客户端页面的话，则可以将exceptionAttribute的值设为null。
- **mappedHandlers**和**mappedHandlerClasses**。如果不为mappedHandlers和mappedHand-lerClasses指定属性的值，那么SimpleMappingExceptionResolver将捕获和处理所有的Handler抛出的异常。可以通过mappedHandlers或者mappedHandlerClasses属性来限定这种默认行为，比如，只指定对某几个Handler或者某几种Handler类型所抛出的异常进行处理。

① 使用文字可能无法清楚地描述这种字符串局部匹配的工作方式，如果你想进一步了解该部分的逻辑，不妨自己读一下SimpleMappingExceptionResolver的代码。

有关SimpleMappingExceptionResolver的各个属性的详细情况可以参考该类的Javadoc。而最后我们要提一下SimpleMappingExceptionResolver的order属性，该属性能够带给SimpleMapping-ExceptionResolver什么特性，我想大家应该可以猜个“八九不离十”了。在Spring MVC框架中，我们可以按照优先级顺序指定多个HandlerMapping以及ViewResolver的实例来帮助我们细化相应关注点的处理，而HandlerExceptionResolver则是框架内第三个拥有这种能力的“人”！也就是说，如果我们在DispatcherServlet的WebApplicationContext中指定多个HandlerExceptionResolver实例的话，DispatcherServlet将根据它们的优先级顺序选取合适的实例进行异常处理。当然啦，优先级的控制方式依然是通过Ordered接口来进行，如代码清单25-15所示。

代码清单25-15　指定多个HandlerExceptionResolver的优先级的配置代码示例

```
<bean name="handlerExResolver1" ➥
class="org.springframework.Web.servlet.handler.SimpleMappingExceptionResolver">
    <property name="order" value="1"/>
    ...
<bean>

<bean name="handlerExResolver2" ➥
class="org.springframework.Web.servlet.handler.SimpleMappingExceptionResolver">
    <property name="order" value="2"/>
    ...
<bean>
...
```

名称依然不重要，DispatcherServlet将依然通过类型来获取和使用容器中所指定的多个HandlerExceptionResolver实例。在使用多个HandlerExceptionResolver实例的时候，有个问题需要注意：如果通过mappedHandlers或者mappedHandlerClasses属性为SimpleMappingExceptionResolver指定一组目标Handler，那么，SimpleMappingExceptionResolver的exception-Mappings和defaultErrorView将只属于这一组指定的目标Handler。如果这之外的Handler抛出异常，那么当前SimpleMappingExceptionResolver将把“控制权”转给下一个。可是如果没有指定mappedHandlers或者mappedHandlerClasses，那么exceptionMappings和defaultErrorView将应用于所有的Handler。这个时候，本来可能不想让当前SimpleMappingExceptionResolver处理的异常类型被抛出，而当前SimpleMappingExceptionResolver自身又找不到合适的处理匹配结果，那么它就会以defaultErrorView作为最后的救命稻草，返回以defaultErrorView作为视图的ModelAndView。如果当前HandlerExceptionResolver具有较高的优先级，那么即使较低优先级的HandlerExceptionResolver中exceptionMappings有对应当前抛出异常类型的处理映射关系，也绝对没有机会“崭露头脚”了。

25.5　国际化视图与LocalResolver

网络拉近了人与人之间的距离。即使相距千里，人们也可以通过网络互相了解对方的信息和文化。但是，不管怎么说，在“地球村”没有统一的“官方语言”之前，不同地区的不同语言依然是人们能够互相交流的一道障碍。所以，现在的Web应用程序尤其是企业级的应用，都会提供国际化的信息支持，以便可以根据访问者的Locale信息为他们提供相应语言的信息内容。为用户提供国际化视图支持自然成为Spring MVC框架不可或缺的一部分。

在ViewResolver根据逻辑视图名解析视图的时候，ViewResolver的resolveViewName

(viewName, locale)方法除了接受要解析的逻辑视图名作为参数之外，还同时接受一个Locale类型对象。这样，ViewResolver就可以根据Locale的不同而返回针对不同Locale的视图实例。到此为止，好像没有必要再往下看了，ViewResolver的设计已经足以完成国际化视图支持的使命了，不是吗？难道ResourceBundleViewResolver不就是很好的例证吗？实际上，从ViewResolver这个层次上来讲，情况确实如此。但是，我们可曾想过，ViewResolver所接受的Locale实例是从何而来的呢？如何获取用户所对应的Locale呢？只有揭开这一谜团，才能将Spring MVC框架内对国际化视图的支持讲述完整。

可以有多种方式获取用户通过浏览器提交的Web请求所对应的Locale值，比如，根据HTTP的Accept-Language协议头进行解析，或者读取用户浏览器端存储的相应Cookie值等。鉴于有如此多不同的处理方式，Spring MVC使用org.springframework.web.servlet.LocaleResolver接口定义对各种可能的Locale值的获取/解析方式进行统一的策略抽象。该接口定义如下：

```
public interface LocaleResolver {
    Locale resolveLocale(HttpServletRequest request);
    void setLocale(HttpServletRequest request, HttpServletResponse response, Locale ➥
    locale);
}
```

作为策略接口，LocaleResolver主要完成两个工作：第一，由Locale resolveLocale (request)方法负责根据当前Locale解析策略获取当前请求对应的Locale值；第二，如果当前策略支持Locale的更改，那么可以通过setLocale(..)方法对当前策略默认取得的Locale值进行变更。

25.5.1 可用的 LocaleResolver

根据通常的Locale获取策略，Spring MVC为LocaleResolver提供了相应的可用实现类，如下所述。

- **FixedLocaleResolver**。最简单的LocaleResolver实现类。一旦指定给FixedLocaleResolver一个Locale值，FixedLocaleResolver将一直持有并返回这个Locale保持不变。因为该FixedLocaleResolver的策略是保持一个Locale值不变，所以不能通过setLocale(..)更改FixedLocaleResolver默认返回的Locale值。
- **AcceptHeaderLocaleResolver**。用户通过客户端浏览器提交Web请求之后，HTTP的Accept-Language协议头（HTTP Header）将随同Web请求一同发送给服务器端进行处理。AcceptHeaderLocaleResolver的策略就是根据Accept-Language协议头来解析并返回当前Web请求对应的Locale值。既然我们无法更改Accept-Language协议头，那么AcceptHeaderLocaleResolver与FixedLocaleResolver一样无法更改默认策略返回的Locale值。
- **SessionLocaleResolver**。SessionLocaleResolver将根据指定键值从Session中获取相应的Locale。初始的时候，我们可以为其指定一个默认的Locale值。如果SessionLocaleResolver既无法从Session获取可用的Locale值，又没有初始化的默认Locale，那么它将采用AcceptHeaderLocaleResolver的策略获取Web请求对应的Locale值。因为SessionLocaleResolver是以Session进行Locale管理，所以我们可以对SessionLocaleResolver默认所返回的Locale值进行变更。
- **CookieLocaleResolver**。如果客户端浏览器没有禁止使用Cookie的话，我们也可以使用Cookie来管理Locale信息。CookieLocaleResolver通过读取客户端的指定Cookie获取相应的Locale值，当然，我们在初始之初就可以为CookieLocaleResolver指定一个默认返回的Locale

值。当CookieLocaleResolver无法从客户端的Cookie获取相应的Locale的时候，它可以转而返回这个初始化时候指定的默认Locale值。如果以上尝试均告失败，那么CookieLocaleResolver也就不得不与SessionLocaleResolver那样，转而使用AcceptHeaderLocaleResolver的策略来获取Locale值了。只要客户端浏览器不禁止Cookie的使用，我们就可以对Cookie中的数据进行更新。所以CookieLocaleResolver支持通过setLocale(..)方法更改默认返回的Locale值。

以上实现类全部位于org.springframework.web.servlet.i18n包中。我们可以根据当前Web应用程序的需要选择使用其中任何一个，有关各个实现类的API细节，不妨参照相应类的Javadoc。

25.5.2 LocaleResolver 的足迹

要在Spring MVC应用中使用相应的LocaleResolver对Locale进行解析和设置，只需要将相应实现类添加到DispatherServlet的WebApplicationContext中。在合适的时机，该LocaleResolver将被使用。我们以SessionLocaleResolver为例，给出对应的配置内容如下：

```
<bean   id="localeResolver"
    class="org.springframework.Web.servlet.i18n.SessionLocaleResolver"
    p:defaultLocale="zh_CN">
</bean>
```

在把要使用的LocaleResolver实现类添加到容器的过程中需要注意，名称“localeResolver”是必须的。因为DispatcherServlet在初始化的时候，将按照该指定名称到WebApplicationContext中去查找可用的LocaleResolver实例。如果找不到，DispatcherServlet将使用默认的FixedLocaleResolver。

如果只使用LocaleResolver，那么将相应实现类添加到WebApplicationContext之后我们就可以收手了。不过，要是想进一步了解添加到WebApplicationContext的LocaleResolver实例都可以在Web请求处理过程中的哪些时间点发挥作用，LocaleResolver走过的几个点还是需要知道一下的。

- 在DispatcherServlet要处理接收到Web请求之前，它会将其在初始化的时候获取的LocaleResolver实例，以及通过该LocaleResolver解析后的Locale值，以LocaleContext的形式绑定到当前线程。这样，在需要的时候，后继处理流程就可以通过绑定的LocaleContext获得当前Locale值以及使用的LocaleResolver。
- 流程按照我们之前所描述的顺序执行，在ViewResolver行动之前，DispatcherServlet将使用初始化时获取到的LocaleResolver进行Locale的解析，然后，ViewResolver就可以使用该LocaleResolver所返回的Locale进行视图查找了。到这里，我们就明白了ViewResolver是如何获得客户端请求对应的Locale值了。
- ViewResolver从“宏观上”解决了对应不同Locale的视图选取问题。不过，如果我们要在具体的视图内访问Locale信息该怎么办？原则上来说，我们是通过org.springframework.web.servlet.support.RequestContext访问必要的国际化信息，包括当前请求对应的Locale、应用使用的LocaleResolver，以及MessageSource中的国际化信息。如果使用JSP作为视图技术，那么直接使用Spring MVC提供的自定义Tag就可以，比如<spring:messagecode=".."/>，因为它底层就是通过RequestContext完成相应功能的。但是，如果使用Velocity/Freemarker之类的视图技术，我们就不得不直接使用RequestContext来完成这些相关信息的访问了。而通过设置AbstractView的requestContext-

Attribute属性可以让我们在这些视图中获取到RequestContext的支持。

- 最后一步是清理的工作，DispatcherServlet将恢复以LocaleContext形式绑定到当前线程的Locale相关信息。

现在，跟着LocaleResolver的足迹走过一遍之后，各位是否已经理解了LocaleResolver的存在价值了呢？

25.5.3 Locale 的变更与 LocaleChangeHandler

当访问各种支持国际化信息页面的网站的时候，即使本地的默认Locale使得服务器返回的是英文的信息页面，我们依然可以点击页面中的相应链接更改这一结果，比如，点击“中文版”切换到中文信息页面。在基于Spring MVC的Web应用中，我们要如何实现这一功能呢？

我们已经介绍了4种LocaleResolver的策略实现，为FixedLocaleResolver、Accept-HeaderLocaleResolver、SessionLocaleResolver和CookieLocaleResolver。前两种实现显然不支持Locale的变更，所以，如果要实现根据用户选择来切换Locale这样的功能需求，我们只能选择SessionLocaleResolver或者CookieLocaleResolve。在这样的前提下，我们再寻求下一步的解决方案。

国际化信息页面的选择是由ViewResolver所接受的Locale决定的。要让用户能够变更到其他语言内容的信息页面，我们只要根据用户提交的请求内容变更Locale值即可。在介绍HandlerInterceptor的时候，我们提到LocaleChangeInterceptor，而这里就是它的“用武之地”了。

LocaleChangeInterceptor的工作原理十分简单，它根据某一个请求参数获取要切换到的Locale信息（该参数默认名称为“locale”），然后通过相应LocaleResolver实现类的setLocale(..)方法，使用新获取的Locale信息替换掉所使用的LocaleResolver默认策略所返回的Locale值。要根据用户请求进行面向不同Locale的视图切换，我们只要配置一个LocaleChangeInterceptor对用户请求进行拦截即可，如代码清单25-16所示。

代码清单25-16 LocaleChangeInterceptor使用配置示例

```
<bean id="handlerMapping" ➥
class="org.springframework.Web.servlet.handler.BeanNameUrlHandlerMapping">
    <property name="interceptors">
        <list>
            <ref bean="marketAccessInterceptor"/>
            <ref bean="localeChangeInterceptor"/>
        </list>
    </property>
</bean>

<bean id="localeChangeInterceptor" ➥
class="org.springframework.Web.servlet.i18n.LocaleChangeInterceptor">
    <property name="paramName" value="lang"/>
</bean>

<bean id="localeResolver" ➥
class="org.springframework.Web.servlet.i18n.CookieLocaleResolver" ➥
p:defaultLocale="en_US">
</bean>
```

如果我们以类似http://host:port/simplefx/anyAction.do?lang=zh_CN，或者http://host:port/simplefx/anyAction.do?lang=nl 这样的请求形式发送处理请求，那么所期望的视图页面就会展现在

眼前了。因为我们配置LocaleChangeInterceptor时使用lang作为标志参数。如果你认为使用默认标志参数即可，那么不要对LocaleChangeInterceptor的paramName属性做任何配置。这样，http://host:port/simplefx/anyAction.do?locale=zh_CN形式的请求将完成同样的功能。

25.6 主题（Theme）与 ThemeResolver

不管是使用Windows操作系统还是使用Linux操作系统，当我们对某种风格的桌面主题感到厌烦的时候，我们就会安装并切换到某种新的桌面主题上。对于Web应用程序来说，为了能够给用户提供更丰富的交互体验，也同样可以提供类似桌面主题的功能。实际上，不管是什么场景下的主题（Theme）功能，它们在本质上都是类似的，无非就是变更一下显示的材质风格：

- 对于操作系统的桌面主题，可能是鼠标样式或者工具条颜色等变更一下；
- 对于Web应用程序来说，可能就是对影响整体风格显示的背景图片，或者某些固定部位的颜色做一些变更。

这就好像我们人穿衣服一样，每天换上不同风格式样的衣服，实际上就是在变换主题啦！

Spring MVC框架提供了对Web应用程序所需要的主题功能的支持，下面具体介绍完成这一功能的几位角色。

25.6.1 提供主题资源的 ThemeSource

通常，Web应用程序的主题是由一些能够影响整体应用显示的静态资源组成的，比如固定位置的背景图片、能够影响页面显示风格的CSS（层叠样式表）文件等。在Spring MVC中，org.springframework.ui.context.ThemeSource负责管理针对各个主题的那些静态资源，该接口定义如下：

```
public interface ThemeSource {
    Theme getTheme(String themeName);
}
```

ThemeSource可以根据指定的主题名称（themeName）查找并返回对应的Theme实例。这样，客户端就可以使用Theme实例中的相应资源来定制视图的显示了。为了能够通过ThemeSource获取相应的主题资源，DispatcherServlet会在处理Web请求之前获取可用的ThemeSource实例，以便能够根据客户端的请求返回相应的主题资源。而实际上DispatcherServlet获取的ThemeSource实例就是它自身所使用的WebApplicationContext。因为WebApplicationContext本身就是一个ThemeSource（它自己实现了ThemeSource接口）。事情到这里并没有完，虽然WebApplicationContext身为ThemeSource，但它和它的实现类一概不干实事。当有主题相关的请求需要处理的时候，它们都是将工作委派给某个ThemeSource的具体实现类，比如org.springframework.ui.context.support.ResourceBundleThemeSource。

ResourceBundleThemeSource允许我们以properties文件来定义每个主题所持有的各项资源，比如：

```
# default.properties
theme.background.image=../images/default-bg-image.jpg
theme.css= ...
...

# blue.properties
theme.background.image=../images/blue-bg-image.jpg
```

```
theme.css=...
...
```

我们分别在default.properties和blue.properties中定义default和blue两个主题对应的材质资源。这样，在具体视图中，我们就可以根据这些主题资源文件中的代码，查找相应的主题资源并应用到视图。比如，如果我们使用JSP作为视图技术，那么可以直接使用Spring提供的theme自定义标签对主题资源进行访问，如下所示：

```
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
...
<style type="text/css">
<!--
body {
    background-image: url(<spring:theme code="theme.background.image"/>);
}
-->
</style>
...
```

而如果是使用其他视图技术，比如Velocity/Freemarker等，就可以通过为相应视图公开RequestContext的方式来访问主题的相关信息。代码清单25-17给出的是Velocity视图模板对应的情况。

代码清单25-17　在Velocity试图模板中访问主题相关信息的代码示例

```
# in velocity template file
<html>
...
<style type="text/css">
<!--
body {
    background-image: url(${rc.getThemeMessage("theme.background.image")});
}
-->
</style>
...
```


不管怎么样，只要将当前使用的主题名称告知ResourceBundleThemeSource，它就能返回对应主题名的properties文件中的相应资源。因为ResourceBundleThemeSource是基于Java标准的ResourceBundle构建的，所以它同样也支持不同Locale下的主题。比如，同样是blue主题，在默认Locale下所使用的主题资源与在其他Locale下所使用的主题资源就可能不同，那么，我们可以按照ResourceBundle国际化支持规则，提供同一主题不同Locale下的主题资源定义，如下所示：

```
# blue.properties
theme.background.image=../images/blue-bg-image.jpg
theme.css=...
...

# blue_zh_CN.properties
theme.background.image=../images/blue-bg-image-with-chinese-characters.jpg
theme.css=...
...
```

现在，如果用户对应中文的Locale并且选择使用blue这一主题的话，ResourceBundleThemeSource将从blue_zh_CN.properties资源文件中为其返回对应的主题资源。

为了能让DispatcherServlet获取到ResourceBundleThemeSource的支持，我们需要将某一

ResourceBundleThemeSource实例添加到DispatcherServlet的WebApplicationContext中，如下所示：

```
<bean   id="themeSource"
    class="org.springframework.ui.context.support.ResourceBundleThemeSource"
    p:basenamePrefix="cn.spring21.simplefx.resources.themes.">
</bean>
```

所配置的ResourceBundleThemeSource实例的bean定义名称必须是“themeSource”，因为默认情况下，WebApplicationContext就是将所有主题相关的请求处理委派给拥有这一名称的ThemeSource实例。默认情况下，ResourceBundleThemeSource将根据主题名称到classpath的根路径下查找相应的properties文件，这当然就要求我们将所有主题properties资源文件放到classpath的根路径下。不过，我们可以通过其basenamePrefix属性定制查找起始路径，就像我们的代码示例所演示的那样，唯一需要注意的就是前缀需要以“.”结束。

在DispatcherServlet有了可用的ThemeSource之后，就会把它绑定到HttpServletRequest的属性上，等着后面谁来用了。那么到底是谁来用呢？

25.6.2　管理主题的 ThemeResolver

现在，通过指定的主题名称，我们就能够从DispatcherServlet所使用的ThemeSource那里获取主题对应的各项资源，然后视图就能够根据这些主题资源来定制视图显示。ThemeSource已经准备就绪了，那主题名称又该如何定夺呢？显然，我们得通过某种方式获取用户当前所选择的主题才行，否则，我们怎么知道使用哪个主题名称到ThemeSource查找要用的主题资源呢？

为了获取并管理用户的Locale信息，Spring MVC提供了LocaleResolver。与此类似，为了获取并管理用户所选择的主题，Spring MVC提供了ThemeResolver。org.springframework.web.servlet.ThemeResolver的主要工作就是解析并获取对应当前请求的主题是什么。如果相应实现机制支持存储的话，也允许对当前请求相关的主题进行设置变更。ThemeResolver的定义如下：

```
public interface ThemeResolver {
    String resolveThemeName(HttpServletRequest request);
    void setThemeName(HttpServletRequest request, HttpServletResponse response, ➥
    String themeName);
}
```

有LocaleResolver在先，我想ThemeResolver并不难理解。显然，DispatcherServlet通过ThemeResolver的resolveThemeName(..)方法就能够获得用户所选择的主题是什么。那么剩下的工作，当然就是根据这个主题的名称到ThemeSource那里获取相应资源进行显示啦。不过，DispatcherServlet肯定无法直接借助于ThemeResolver这一接口来完成工作，所以，还是来看一下Spring MVC都提供了哪些可用的ThemeResolver实现类吧！

除了不能像LocaleResolver那样通过HTTP的Accept-Language协议头来获取主题信息之外，ThemeResolver可以使用LocaleResolver所使用的其他三种策略来获取并且管理用户的主题，如下所述。

- **org.springframework.web.servlet.theme.FixedThemeResolver**。如果我们不明确为DispatcherServlet指定任何ThemeResolver实例供其使用，DispatcherServlet将默认使用FixedThemeResolver来管理用户的主题。顾名思义，一旦使用FixedThemeResolver指定了主题之后，主题将保持不变。所以，对FixedThemeResolver进行setThemeName(..)操作显然是不行的。

- **org.springframework.web.servlet.theme.SessionThemeResolver** 。 HttpSession 是 SessionThemeResolver得以生存的土壤，SessionThemeResolver将按照指定的属性名称到Session中获取用户的主题。如果找不到，则使用默认的主题。这可以通过其default-ThemeName属性进行指定。因为我们可以对Session的属性进行设置，所以，SessionThe-meResolver可以通过setThemeName(..)方法重新设置用户主题。
- **org.springframework.web.servlet.theme.CookieThemeResolver**。SessionTheme-Resolver以HttpSession作为主题信息的载体，而CookieThemeResolver则以Cookie作为主题信息的载体。只要用户端浏览器不禁止Cookie的使用，我们就可以使用CookieThemeResolver对用户选择的主题进行管理，包括获取和更新。

现在，只要将以上任一ThemeResolver实现添加到DispatcherServlet的WebApplication-Context中，DispatcherServlet就能够“左右逢源”了：

```
<bean id="themeResolver"
    class="org.springframework.Web.servlet.theme.SessionThemeResolver"
    p:defaultThemeName="default">
</bean>
```

我们这里使用了SessionThemeResolver。不管使用哪种ThemeResolver，注册到容器的bean定义的名称为“themeResolver”是必须的，因为DispatcherServlet初始化的时候将根据这一名称到其自己的WebApplicationContext中获取可用的ThemeResolver实例。

25.6.3 切换主题的 ThemeChangeInterceptor

如果用户永远只能使用一种风格的主题，那么显然提供主题的功能就没有了任何的意义。只有允许用户根据喜好切换主题，才能够体现主题功能丰富用户体验的价值。

在Spring MVC中，用户要切换Locale有LocaleChangeInterceptor来相助，而用户要切换主题的时候，也同样有ThemeChangeInterceptor来帮忙。只要将用户选择要切换到的主题以某个参数提交到服务器端处理，ThemeChangeInterceptor就能够根据这一参数重新设置用户所使用的主题，之后，视图就可以获取切换后的主题来定制视图的显示了。当然，要让ThemeChangeInterceptor生效，我们自然需要将它与提交主题变更的Web请求联系到一起，如代码清单25-18所示。

代码清单25-18 ThemeChangeInterceptor使用配置示例

```
<bean id="handlerMapping" ➥
class="org.springframework.Web.servlet.handler.BeanNameUrlHandlerMapping">
    <property name="interceptors">
        <list>
            <ref bean="themeChangeInterceptor"/>
            <ref bean="marketAccessInterceptor"/>
            <ref bean="localeChangeInterceptor"/>
        </list>
    </property>
</bean>

<bean id="themeChangeInterceptor" ➥
class="org.springframework.Web.servlet.theme.ThemeChangeInterceptor">
</bean>
...
```

ThemeChangeInterceptor默认以名称为“theme”的参数作为要切换的主题名称。比如，以

http://host:port/simplefx/anyrequest.do?theme=blue形式发送的Web请求，将最终被切换到blue主题显示风格。如果我们不想使用“theme”作为标志参数，那么可以通过设置ThemeChangeInterceptor的paramName属性变更这一默认使用的标志参数。

25.7　小结

至此，我们基本上已经认识了Spring MVC家族的所有成员，包括MultipartResolver、HandlerAdaptor、HandlerInterceptor、HandlerExceptionResolver、LocalResolver以及ThemeResolver，也了解了这些家族成员之间的关系和它们各自的功能。应该说，到此为止，整个Spring MVC的介绍就该打住了。但是，各种新的技术理念层出不穷，为了能够进一步简化基于Spring MVC的开发，Spring团队从来没有停下追求新理念的脚步。接下来，我们将看到Spring MVC框架是如何在现有框架结构的基础上支持各种新的开发理念的，首先映入眼帘的，则是Spring2.5发布后新引入的基于注解的Controller。

第26章

Spring MVC中基于注解的Controller

本章内容

- 初识基于注解的Controller
- 基于注解的Controller原型分析
- 近看基于注解的Controller

随着Java5-Tiger的发布，注解越来越受到开发人员的喜爱。顺应形势，Spring MVC在Spring 2.5发布中新添加了一种基于注解的Controller形式。借助于与Spring 2.5一同发布的容器内<context:component-scan>功能支持，基于注解的Controller几乎可以达到XML零配置，进而极大地提高我们的开发效率。

26.1 初识基于注解的Controller

在Spring MVC框架中，传统的Handler类型，比如Controller或者ThrowawayController，都需要具体实现类继承某个基类或者实现某个接口。而使用基于注解的Controller的话，则没有这样的限制。实际上，基于注解的Controller就是一个普通的POJO，只是使用某些注解附加了一些相关的元数据信息而已。下面是某个基于注解的Controller实现：

```
@Controller
@RequestMapping("/helloAnnoController.anno")
public class AnyTypeYouLikeController {
    @RequestMapping(method={RequestMethod.GET,RequestMethod.POST})
    public String processWebRequest()
    {
        return "anno/helloAnnoController";
    }
}
```

可以看到，我们的Controller实现类AnyTypeYouLikeController无拘无束，不需要实现任何强制接口类型，或者去继承哪个父类。最主要的是也不需要去依赖Servlet API，甚至Spring MVC相关的API。AnyTypeYouLikeController只作为一个普通的POJO而存在。不过，应用程序中类似的POJO到处可见，如果不能通过某种方式加以区分，Spring MVC显然无法知道，到底哪个POJO才是用于Web请求处理的Controller实现类。纵使要“大海捞针”，我们也得知道“针”到底是个什么样子不是？

传统的Controller或者ThrowawayController通过接口类型作为标志的方式，基于注解的

Controller则采用标注于具体实现类上的某种类型的注解作为标志。在AnyTypeYouLikeController类的定义中，我们使用@Controller和@RequestMapping两种类型的注解来标注该类，以告知Spring MVC框架AnyTypeYouLikeController可以作为处理某一Web请求的Controller实现类。原则上来说，只需要在WebApplicationContext中添加如下配置：

```
<context:component-scan base-package="your.controller.package"/>
```

然后将各种基于注解的Controller实现类定义在your.controller.package下即可。剩下的事情，像如何获取并调用这些基于注解的Controller等，就由Spring MVC框架帮我们全部揽下来了。对于/helloAnnoController.anno形式的请求，Spring MVC将使用AnyTypeYouLikeController作为处理请求的Handler，并调用其标注了@RequestMapping的processWebRequest()方法进行当前Web请求的处理。

怎么样？较之传统的Handler类型，使用基于注解的Controller是不是看起来要简洁得多？基本上，只要在指定的包下面定义用于处理Web请求的Handler对象（任何你我喜欢的对象类型），然后使用指定的注解类型标注它们就行，完全省却了在Java文件与XML配置文件之间切换的烦恼。最主要的是，这些基于注解的Controller甚至不依赖于Servlet API。而且，你或许还能找出更多喜爱它的理由。不得不承认，对于日常开发来说，事情确实简单了不少。但是，难道你就不想一探基于注解的Controller之下到底隐藏着什么样的秘密吗？如果没有Spring MVC框架幕后的某种支持，你想啊，孤伶伶的一个使用某种注解类型标注的POJO，又能够发挥什么样的作用呢？

26.2　基于注解的Controller原型分析

在我们完全明白Spring MVC框架的整个结构之后，添加个基于注解的Controller已经不再是什么高难度动作了。对于Spring MVC框架来说，基于注解的Controller和传统的Controller或者ThrowawayController在本质上并没什么区别，它们全都是框架内用于处理Web请求的Handler。如果我们可以达成这样的共识，那么理解基于注解的Controller的运作机制就并非难事了。

基于注解的Controller是Spring 2.5之后才出现的Handler形式。我们不妨先让自己回到“史前文明”，看一下要在Spring 2.5之前版本的Spring MVC中使用基于注解的Controller需要做哪些工作，问题实际上就简化为，如何实现自定义的Handler类型，对吗？

显然，基于注解的Controller这种类型的Handler定义我们已经有了，只是一些添加了某种类型注解标注的POJO而已，下面要做的是要搞清楚如下两个问题。

- 如何让Spring MVC框架类（其实就是DispatcherServlet）知道当前Web请求应该由哪个基于注解的Controller处理？
- 如何让Spring MVC框架类知道调用基于注解的Controller的哪个方法来处理具体的Web请求？

用我们的行话来说就是，我们需要为基于注解的Controller提供相应的HandlerMapping以处理Web请求到Handler的映射关系，需要提供相应的HandlerAdaptor以调用并执行自定义handler的处理逻辑。如果没记错的话，在25.2节，我们已经提到自定义Handler的时候需要考虑到这两点。

26.2.1　自定义用于基于注解的Controller的HandlerMapping

在基于注解的Controller中，当前实现类将用于哪个Web请求的处理是由相应的注解标注的。比如，@RequestMapping("/helloAnnoController.anno")标注的AnyTypeYouLikeController将用于URL为“/helloAnnoController.anno”的Web请求处理。这些注解中的信息，我们需要通过Java

的反射机制（Java Reflection）来读取。无论是现有的BeanNameUrlHandlerMapping还是SimpleUrlHandlerMapping，显然都没有提供通过反射读取注解中映射信息的功能。所以，我们不得不为基于注解的Controller提供特定的HandlerMapping实现。

要实现一个专门用于基于注解的Controller的HandlerMapping实现，从原理上来说并不困难。我们所要作的只是遍历所有可用的基于注解的Controller实现类，然后根据请求的路径信息，与实现类中的注解所标注的请求处理映射信息进行比对。如果当前基于注解的Controller实现类的注解所包含的信息与请求的路径信息相匹配，那么就返回当前这一基于注解的Controller实现类即可。所以，代码清单26-1所示的原型代码也就构建出来了。

代码清单26-1　根据注解信息提供Web请求映射功能的HandlerMapping原型代码示例

```
public class AnnotationBasedHandlerMapping implements HandlerMapping {
    private List<HandlerInterceptor> handlerInterceptors;

    public HandlerExecutionChain getHandler(HttpServletRequest request) ➥
    throws Exception {
        HandlerExecutionChain chain = null;

        Object[]AnnotationControllers = getAvailableAnnotationControllers();
        for(Object Annotation Controller: AnnotationControllers)
        {
            Class<?> clazz = AnnotationController.getClass();
            if(clazz.isAnnotationPresent(RequestMapping.class))
            {
                RequestMapping mapping = clazz.getAnnotation (RequestMapping.class);
                if(matches(mapping,request))
                {
                chain = new HandlerExecutionChain(AnnotationController);
                if(!CollectionUtils.isEmpty(getHandlerInterceptors()))
                    chain.addInterceptors(getHandlerInterceptors().toArray(new ➥
                    HandlerInterceptor[getHandlerInterceptors().size()]));
                    break;
                    }
            }
        }
        return chain;
    }

    protected Object[] getAvailableAnnotationControllers() {
        // TODO
    }
    protected boolean matches(RequestMapping mapping, HttpServletRequest request) {
        // TODO
        }

    public List<HandlerInterceptor> getHandlerInterceptors() {
     return handlerInterceptors;
      }

      public void setHandlerInterceptors(List<HandlerInterceptor> handlerInterceptors)
      {
      this.handlerInterceptors = handlerInterceptors;
      }
}
```

在HandlerMapping接口必须实现的getHandler(..)方法中，我们首先遍历了所有可用的基

于注解的Controller，然后通过反射获取@RequestMapping的相应信息与请求信息进行匹配，并最终返回匹配后的结果。这一总体逻辑很好理解，我们可能存在疑问的地方是后面的两个辅助方法，即getAvailableAnnotaitionControllers()和matches(..)。

getAvailableAnnotaitionControllers ()方法用于获取所有基于注解的Controller实现类。如果你愿意，当然可以把所有的这些标注了注解的Controller实现类添加到WebApplicationContext，然后注入给我们的AnnotationBasedHandlerMapping，并在getAvailableAnnotationControllers()方法中直接返回。不过，使用基于注解的Controller的一个优势就在于，不需要在WebApplicationContext中添加任何配置。所以，即使能这么做，我们也不应如此。不妨将这个问题暂且搁下，稍后我们再来揭示Spring 2.5中是如何处理它的，目前我们只要了解这个方法是做什么的就可以。

至于matches(..)方法，其实现逻辑完全就是一个获取@RequestMapping所包含的信息，然后与当前请求进行对比的过程，这可以借助于某些框架类进行匹配，比如org.springframework.util.PathMatcher和org.springframework.web.util.UrlPathHelper等。当然，如果你不辞辛苦，完全凭一己之力去写所有匹配代码来完成这一工作也是可以的。

现在，只要将我们的AnnotationBasedHandlerMapping添加到DispatcherServlet的字距WebApplicationContext，理论上它就能正确处理Web请求到具体基于注解的Controller之间的映射关系。但原型终归是原型，它只是帮助我们简化问题的难度，从而了解一件事情本质上的东西。要应用于实际环境还有许多路要走，好在，这段路已经有人走过了。

实际上，正如我们所预料的那样，Spring 2.5中基于注解的Controller确实依赖于某一个官方的HandlerMapping实现来处理映射关系，那就是org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping。DefaultAnnotationHandlerMapping在实现原理上与我们的AnnotationBasedHandlerMapping原型相似，它会首先扫描应用程序的Classpath，通过反射获取所有标注了@Controller的对象，之后就可以像AnnotationBasedHandlerMapping所展示的那样来完成一个HandlerMapping的职责了。至于扫描Classpath然后获取标注了@Controller的对象这样的工作，是由<context:component-scan/>来完成的。这也就是为什么需要我们的基于注解的Controller实现类必须标注@Controller类型的这一注解的原因。因为<context:component-scan/>内置了对@Controller的支持，但并不一定内置了其他你所需要的注解类型的支持。如果你不想使用@Controller，那么就得自己实现扫描classpath并获取相应Controller对象的逻辑。不过那又何苦呢？当然，不管你最终如何选择，我想，看到这里，如何实现原型中的getAvailableAnnotationControllers()方法你已经心中有数了吧？

> **注意**　DefaultAnnotationHandlerMapping实现逻辑要比AnnotationBasedHandlerMapping原型复杂得多。随着后面对基于注解的Controller特性的深入，我们就会认识到这一点。

在2.5版本的Spring MVC中，DefaultAnnotationHandlerMapping将在DispatcherServlet初始化的时候就被默认启用（随同一起的还有BeanNameUrlHandlerMapping）。除非想要定制它的某些行为，大多情况下，并不需要在DispatcherServlet的WebApplicationContext中明确地声明它。

26.2.2　自定义用于基于注解的 Controller 的 HandlerAdaptor

有了针对基于注解的Controller的HandlerMapping实现，只是完成了一半的工作。要让基于注

解的Controller真正工作起来，我们还需要完成剩下的那一半，即实现针对基于注解的Controller的自定义HandlerAdaptor。

HandlerMapping返回了处理Web请求的某个基于注解的Controller实例，而DispatcherServlet并不知道应该去调用该实例的哪个方法来处理Web请求。为了能够以统一的方式调用各种类型的Handler，DispatcherServlet需要一个针对基于注解的Controller的HandlerAdaptor实现。这跟"传统的Controller需要一个SimpleControllerHandlerAdapter，ThrowawayController需要一个ThrowawayControllerHandlerAdapter"是一个道理。

为了构建一个针对基于注解的Controller的HandlerAdaptor原型实现，我们得回头看一下基于注解的Controller所定义的Web请求处理方法有何特征。我们知道，对于传统的Controller来说，SimpleControllerHandlerAdapter只需要调用其接口中所定义的handleRequest(..)方法即可。对于ThrowawayController情况也是相似的，ThrowawayControllerHandlerAdapter知道execute()方法就是ThrowawayController所定义的Web请求处理方法，而基于注解的Controller显然无法找到这样的契约关系。不过，我们依然可以找到特定于它的Web请求处理方法，那就是使用@RequestMapping标注的方法定义。（@RequestMapping可以有两种用途，我们稍后详述。）

与DefaultAnnotationHandlerMapping（或者我们的AnnotationBasedHandlerMapping）能够使用反射获取相应注解的信息这一做法类似，针对基于注解的Controller的HandlerAdaptor原型实现，同样可以如法炮制，只要通过反射查找标注了@RequestMapping的方法定义，然后通过反射调用该方法，并返回DispatcherServlet所需要的ModelAndView即可。有了这样的思路，代码清单26-2所示的一个HandlerAdaptor原型也就不难构建了。

代码清单26-2　对应基于注解的Controller的HandlerAdaptor原型代码示例

```
public class AnnotationControllerHandlerAdaptor implements HandlerAdapter
{
    public boolean supports(Object handler) {
        Class<?> clazz = handler.getClass();
        return clazz.isAnnotationPresent(Controller.class) || ➥
        clazz.isAnnotationPresent(RequestMapping.class);
    }

    public ModelAndView handle(HttpServletRequest request, ➥
    HttpServletResponse response, Object handler) throws Exception {
        Method[] methods = handler.getClass().getDeclaredMethods();
        for(Method method : methods)
        {
            if(method.isAnnotationPresent(RequestMapping.class))
            {
                ModelAndView mav = invokeAndReturn(method,handler,request);
                return mav;
            }
        }
        response.sendError(HttpServletResponse.SC_NOT_FOUND);
        return null;
    }

    private ModelAndView invokeAndReturn(Method method, Object handler, ➥
    HttpServletRequest request) {
        // 1.使用DataBinder或者其他装备将request参数绑定到方法参数
        Object[] parameterValues = bind(request,method);
```

```
        // 2.使用绑定后获得的相应参数调用方法
        Object returnValue = method.invoke(handler, parameterValues);
        ModelAndView mav = new ModelAndView();
        if(returnValue instanceof String)
            mav.setViewName((String)returnValue);
        else if(returnValue instanceof ModelMap)
            mav.addAllObjects((ModelMap)returnValue);
        else
            ...;
        return mav;
    }

    public long getLastModified(HttpServletRequest arg0, Object arg1) {
        return -1;
    }
}
```

实际上，一个真正用于生产环境下，对应基于注解的`Controller`的`HandlerAdaptor`实现类要考虑的事情很多。而我们的原型实现因为要将问题简化，所以并没涉及，如下所述。

- 如何根据`@RequestMapping`提供的各种信息来决定是否调用当前方法？我们的原型只要发现当前方法标注有`@RequestMapping`，即认为它就是将被调用的Web请求处理方法，那如果`@RequestMapping`指定了`method={RequestMethod.POST}`，我们同样可能在Web请求以GET形式发送的时候调用当前方法，这显然是违反`@RequestMapping`语义的。
- 如何在数据绑定期间决定将哪个请求参数绑定到方法的哪个参数上？通过现有的反射API可以获取当前方法的参数类型，但无法获取方法参数的名称，单靠反射API显然无法识别请求参数到方法参数一对一的映射关系，也就无法实现正确的数据绑定。为了解决这个问题，Spring 2.5在实现类似的功能的时候，使用了ObjectWeb的ASM（http:// asm.objectweb.org/）类库帮助解决方法参数名称的获取问题。
- 方法参数同样可能持有相应的注解，显然需要为这些标注有相应注解的方法参数的处理提供更多的分支逻辑，以囊括所有可能的情况，比如，如果指定Web请求参数中的author需要绑定到处理方法的name方法参数上去，我们可能需要定义处理方法如下：

```
public String processMethod(@RequestParam("author") String name, ...){
    ...
}
```

这种情况下，我们不能依赖默认的请求参数与方法参数名称匹配的原则进行数据绑定，而应该以`@RequestParam`所指定的绑定原则为准。这就需要我们添加更多的代码逻辑支持。

- 如果基于注解的`Controller`中某些数据需要通过`HttpSession`进行管理，而基于注解的`Controller`又不依赖于任何ServletAPI，该如何在`HandlerAdaptor`中提供代码逻辑支持呢？
- 基于注解的`Controller`的请求处理方法返回值有没有限制？如果有，可以定义哪些类型？如果没有，`HandlerAdaptor`要如何枚举所有可能的返回值类型呢？我们的原型中只是考虑了两种类型。如果有其他的类型，难道要继续枚举未知的类型？在Spring 2.5的基于注解的`Controller`中，处理方法的返回值类型只能有规定的几种，所以，这个问题选择了比较简单的解决方案。
- 如果基于注解的`Controller`需要访问模型数据或者返回某些模型数据，我们的`Handler-`

Adaptor可以通过哪些方式提供支持？是让具体的处理方法通过返回值将模型参数返回，然后由HandlerAdaptor添加到要返回给DispatcherServlet使用的ModelAndView中，还是由HandlerAdaptor实现为基于注解的Controller传递一个能够进行模型数据访问的对象引用？这显然也是需要考虑的。

说了这么多，实际上就一个目的，实现一个针对基于注解的Controller的HandlerAdaptor的目的是明确的，那就是将请求信息绑定到具体的Controller实例，然后调用相应的处理方法，并将返回结果以ModelAndView的形式返回给DispatcherServlet使用。但是，要实现这样一个HandlerAdaptor的工作量和要考虑的关注点是很多的。所以，理解原理就好。如果要使用这样一个HandlerAdaptor实现类，还是使用Spring 2.5提供的官方支持吧！

Spring 2.5中为基于注解的Controller提供的HandlerAdaptor实现类是org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter。该类为我们考虑并实现了以上提到的几乎所有问题，甚至还要更全面。默认情况下，2.5版本的DispatcherServlet将在初始化的时候就实例化了一个AnnotationMethodHandlerAdapter，用于支持基于注解的Controller。我想，如果没有对AnnotationMethodHandlerAdapter的定制需求的话，通常就不用在DispatcherServlet的WebApplicationContext中明确地声明一个该类的bean定义了。

26.3 近看基于注解的 Controller

在初步了解了Spring 2.5提供的基于注解的Controller相关内容之后，让我们开始近距离地接触一下基于注解的Controller的方方面面，以便可以更好地了解它，使用它。

只介绍各种相关概念，难免会枯燥乏味，我们不妨引入一个场景。在这个场景的基础上逐步了解基于注解的Controller相关的各个方面。在我们的FX系统中，无论是面向顾客的报表还是法定报表，按照相关法律规定，都需要存储指定的年限以上，比如至少存储5年。这个期限是可选的，针对每个报表可能情况各异，所以，我们需要一个针对每个报表存储期限的设定功能。下面就将以这一功能的开发为背景，来详细介绍基于注解的Controller的各项特性。开始之前，不妨先看一下最终效果图吧（见图26-1）！

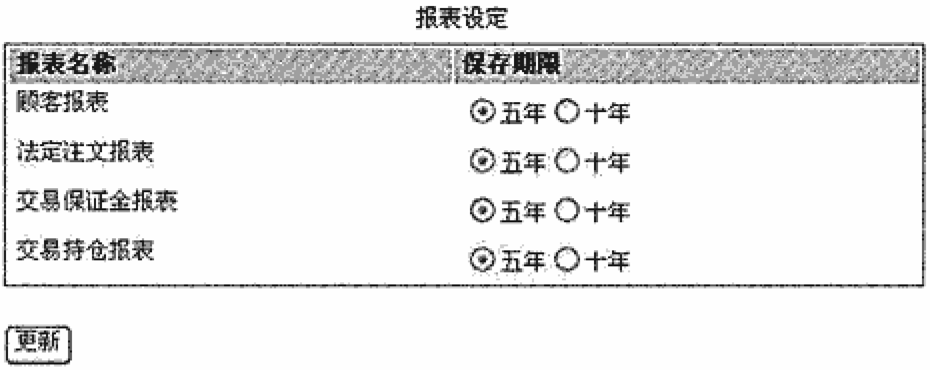

图26-1 报表保存设定的视图效果

我们需要提供一个基于注解的Controller实现，来处理这一视图相关的各个方面。现在马上开始着手……

26.3.1 声明基于注解的 Controller

一个基于注解的Controller实现类，在真正成为一个Controller之前，只是一个普通的POJO。

我们可以先给出一个概念上的对象类定义，如下所示：

```
public class ReportSettingAnnotationController {
    ...
}
```

目前为止，我们只是先假定，它将用于报表设定功能相关的Web请求的处理。为了让它成为一个真正基于注解的Controller，我们还有许多事情要做！

1. 再谈@Controller

要想使一个普通的对象定义成为Spring MVC框架使用的基于注解的Controller，需要用@Controller这个注解进行标注，例如：

```
@Controller
public class ReportSettingAnnotationController {
    ...
}
```

@Controller是基于注解的Controller的身份象征，它主要有如下两个作用。

- 使用<context:component-sacn/>之后，标注了的对象可以被纳入Spring的IoC容器进行管理，这样，就可以为其提供依赖注入等服务。
- 在DefaultAnnotationHandlerMapping查找对应的Controller以处理当前Web请求的时候，可以将标注了@Controller的对象定义列入被考虑的范畴。

所以，对于每一个“基于注解的Controller”的实现类来说，@Controller是必须的！

2. @RequestMapping详解

只拥有@Controller这一个标志性注解，并不能让Spring MVC框架知道，针对当前Web请求应该调用哪个基于注解的Controller实现类。我们需要通过@RequestMapping为DefaultAnnotation-HandlerMapping提供必要的映射信息。

@RequestMapping可以被标注于类型声明或者方法声明上。如果声明在类型定义上，那么@RequestMapping通常作为请求处理映射信息的提供者。如果声明在方法定义上，那么@Request-Mapping更多的是为了表明当前方法定义是一个Web请求处理方法。当然，这并不妨碍@Request-Mapping同时提供请求处理相关的映射信息。鉴于@RequestMapping的双重身份，我们通常可以通过如下的组合方式来使用它。

- 标注于类型定义的**@RequestMapping**提供映射信息，标注在方法定义的**@RequestMapping**标志具体的请求处理方法。为了能够像SimpleFormController那样，同时提供表单的显示和处理功能，我们的ReportSettingAnnotationController就是采用这种方式来声明的，如代码清单26-3所示。

代码清单26-3　@RequestMapping使用代码示例

```
@Controller
@RequestMapping("/reportSetting.anno")
public class ReportSettingAnnotationController
{
    @RequestMapping(method=RequestMethod.GET)
    public String displayReportSettings(..){
        ...
    }
    @RequestMapping(method=RequestMethod.POST)
    public String updateReportSetting(..){
```

```
        ...
    }
}
```

类型定义上标注的@RequestMapping("/reportSetting.anno")，使得所有提交到/reportSetting.anno的Web请求都将由ReportSettingAnnotationController来处理。但是，以GET方法提交的HTTP请求将由@RequestMapping(method=RequestMethod.GET)所标注的displayReportSettings(..)方法来处理，而以POST方法提交的HTTP请求将由@RequestMapping(method=RequestMethod.POST)所标注的updateReportSetting(..)方法来处理。

在方法级别使用@RequestMapping来限定请求的处理范围的时候，可以指定两个属性，除了我们刚使用的method属性，还有一个params属性。使用params属性，可以达到与使用ParameterMethodNameResolver作为MethodNameResolver的MultiActionController类似的功能。下面是我们可以为params属性指定的两种表达式形式。

- **"parameterName=parameterValue"形式的表达式。**表达式的前半部分，即参数名，通常相同，由后面的参数值部分来区分并限定请求的处理范围，如代码清单26-4所示。

代码清单26-4　结合params使用的@RequestMapping代码示例一

```
@Controller
@RequestMapping("/requestUrl.anno")
public class MockMultiActionAnnotationController
{
    @RequestMapping(params="locale=en",method={RequestMethod.POST})
    public String processMethodOne(..)
    {
        ...
    }
    @RequestMapping(params="locale=zh",method={RequestMethod.POST})
    public String processMethodTwo(..)
    {
        ...
    }
}
```

如果通过http:// ../application/requestUrl.anno?parameterName=value的形式发送请求，那么不同的value值将决定当前Web请求由哪个方法定义进行处理。

- **"parameter"形式的表达式。**只为每个@RequestMapping的params提供某个参数名，由请求中是否存在某一参数来决定将当前请求交给哪个方法处理，如代码清单26-5所示。

代码清单26-5　结合params使用的@RequestMapping代码示例二

```
@Controller
@RequestMapping("/requestUrl.anno")
public class MockMultiActionAnnotationController
{
    @RequestMapping(params="delete",method={RequestMethod.POST})
    public void processMethodOne()
    {
        ...
    }
    @RequestMapping(params="update",method={RequestMethod.POST})
    public void processMethodTwo()
    {
```

26

```
        ...
    }
}
```

不管你将使用哪种方式，有一点是要注意的，启用了params的@RequestMapping只允许用于方法定义上。对启用了method属性的@RequestMapping也是同样道理。

- @RequestMapping全部标注于方法定义上，类型定义不标注任何@RequestMapping。如果我们想让同一个基于注解的Controller实现类中不同的方法定义对应不同的Web请求处理，那么可以为每个方法定义都提供一个@RequestMapping，而每个@RequestMapping将提供Web请求处理所需要的所有信息。这种形式的基于注解的Controller定义类似于结合Properties-MethodNameResolver使用的MultiActionController，如代码清单26-6所示。

代码清单26-6　全部标注于方法级别的@RequestMapping使用代码示例

```
@Controller
public class MultiActionAnnotationController
{
    @RequestMapping("/request1.anno")
    public String processMethod1()
    {
        ...
    }

    @RequestMapping(value="/request2.anno",method={RequestMethod.POST})
    public ModelMap processMethod2()
    {
        ...
    }

    @RequestMapping(value="request3.anno",method={RequestMethod.GET,RequestMethod.POST})
    public void processMethod3()
    {
        ...
    }
}
```

现在，MultiActionAnnotationController将可以服务于多个不同的Web请求的处理。

基本上，这两种组合方式已经可以囊括所有可能的情况了。更多信息可以参考该注解定义的Javadoc文档。

3. 请求处理方法签名规则说明

我们使用@Controller标注ReportSettingAnnotationController，使它成为了一个基于注解的Controller，又使用@RequestMapping限定ReportSettingAnnotationController要处理的Web请求，以及该类的哪些方法定义将作为Web请求的处理方法。下一步要搞清楚的，就是如何来定义那些由@RequestMapping标注了的Web请求处理方法。只是使用@RequestMapping标注这些方法，并不能让它们成为你肚子里的蛔虫，我们还是得自己去实现相应的处理逻辑。不过，在这之前，我们得先对用于Web请求处理的方法定义做“约法三章”。

使用@RequestMapping标注的Web请求处理方法的签名比较灵活，我们几乎可以声明并使用任何类型的方法参数。不过，以下几种类型的方法参数将拥有更多语义，它们均来自框架内部（或者说AnnotationMethodHandlerAdapter）所管理的对象引用，通过声明这些类型的方法参数，我们就可以获得请求处理过程中所需要的各种依赖。

request/response/session。只要我们在方法定义中声明一个HttpServletRequest形式的方法参数。如下所示：

```
@RequestMapping(..)
public void processMethod(HttpServletRequest request,..){
    ...
}
```

就可以在该处理方法中直接使用对应当前Web请求的request对象。框架类将保证，该方法被调用的时候会被传入当前的request对象实例。对于response和session也是同样的道理。

org.springframework.web.context.request.WebRequest。与为处理方法声明request/response/session类型的参数效果类似，我们将在当前处理方法中获得可用的WebRequest实例。

java.util.Locale。通过相应LocaleResolver所返回的对应当前Web请求的Locale。

java.io.InputStream/java.io.Reader。用于访问对应当前Web请求的request的内容，相当于声明了HttpServletRequest类型参数后，调用request.getInputStream()或者request.getReader()所获得的对象引用。

java.io.OutputStream/java.io.Writer。它们来自response.getOutputStream()和response.getWriter()，所以，我们可以通过它们向客户端写入相应的数据。

java.util.Map / org.springframework.ui.ModelMap。如果声明了Map或者ModelMap类型的方法参数，那恭喜你，现在可以对模型数据为所欲为了。要获得相应数据也好，添加相应数据也好，这些数据都将反映到视图所对应的模型数据上，例如：

```
@RequestMapping(..)
public void processMethod(ModelMap model)
{
    // 添加模型数据
    model.addAttribute("commandObject",..);
    // 根据Key获取相应模型数据
    Object cmd = model.get("command");
}
```

你或许会很诧异这怎么可能？实际上，这并不难理解啊。如下所示的原型代码或许可以更容易看清楚到底发生了什么：

```
ModelAndView mav = ...;
ModelMap modelMap = new ModelMap();
// 调用
processMethod(modelMap);
// 将模型数据添加到ModelAndView
mav.addAllObjects(modelMap);
return mav;
```

框架类只需要在调用具体处理方法的时候，将事先初始化的Map或者ModelMap实例传给要调用的方法即可。调用完成后，再把其中的数据添加到要返回给视图的ModelAndView就成了。如果你能理解这一点，也就可以触类旁通了。这里所说的这几种特殊类型的方法参数声明得以被特殊对待，基本上都是类似的道理。

使用@RequestParam或者@ModelAttribute所标注的方法参数。相应方法参数将根据这两个注解的语义获得相应的对象值。我们将在介绍@RequestParam和@ModelAttribute的时候进一步说明这个问题。

org.springframework.validation.Errors/org.springframework.validation.BindingResult。用于对Command对象进行数据验证的Errors或者BindingResult对象。我们可以向这两种类型的方法参数查询数据验证的结果，也可以在执行相应Validator进行数据验证之后，向这两种类型的方法参数中添加验证数据，如代码清单26-7所示。

代码清单26-7　BindingResult使用代码示例

```
public String processMethod(@ModelAttribute("command") ReportSettings reportSettings, ➥
BindingResult result)
{
    Validator validator = ...;
    ValidationUtils.invokeValidator(validator,reportSettings,result);
    ...
    if(result.hasErrors())
    {
        return FORM_VIEW_NAME;
    }
    return "other ViewName";
}
```

声明Errors或者BindingResult类型的方法参数有一个限制，它们的声明必须紧跟Command对象定义。在我们的代码示例中，BindingResult声明紧跟在Command对象ReportSettings之后，而其他类型的方法参数声明是没有任何顺序限制的。

org.springframework.web.bind.support.SessionStatus。SessionStatus主要用于管理请求处理之后Session的状态，比如清除Session中的指定数据。在我们的ReportSettingAnnotationController完成之际，SessionStatus将再次浮出水面。

除了以上列举的这几种特殊类型的方法参数声明，如果请求处理方法存在其他类型的方法参数声明的话，框架类会根据方法参数的名称、类型，或者注解标志，将Web请求的相应参数绑定到这些参数上。在详细说明基于注解的Controller中的数据绑定功能之前，我们还是先来看一下Web请求的处理方法都可以返回哪些类型的返回值吧！

基于注解的Controller的请求处理方法返回值类型可以有如下4种形式。

- **org.springframework.web.servlet.ModelAndView**。这个不用多说了吧，视图信息和模型信息都能通过它返回。
- **java.lang.String**。该类型返回值代表逻辑视图名，模型数据需要以其他形式提供，比如为处理方法声明一个ModelMap类型的参数。
- **org.springframework.ui.ModelMap**。ModelMap类型返回值只包含了模型数据信息而没有视图信息，框架类将根据请求的路径，按照默认规则提取相应的逻辑视图名来使用，比如：

  ```
  @RequestMapping("/simple.anno")
  public ModelMap processMethod()
  {
      ...
  }
  ```

  以上方法定义中，simple将被作为逻辑视图名而使用。
- **void**。没有任何返回值，视图信息需要从请求路径中提取默认值，模型数据需要通过其他形式提供。结合只返回String类型和只返回ModelMap类型两种情况下的处理逻辑，也就不难想象框架类将如何处理void类型的返回值了。

总之，为了简化框架类的实现逻辑，处理方法的返回值只允许以上4种形式。开发中将选择哪种类型，完全因人而异。

26.3.2 请求参数到方法参数的绑定

如果基于注解的Controller的处理方法声明的参数类型属于我们规定的那几种特殊类型，那么框架类将为这些特殊类型的方法参数提供框架内管理的相应对象的引用。而如果处理方法所声明的方法参数类型在规定的几种特殊类型之外，那么Spring MVC将根据某些规则把请求参数绑定到这些方法参数上。下面我们要说的就是，如何在基于注解的Controller中实现请求参数到相应的方法参数的数据绑定。

1. 默认绑定行为

默认绑定行为是指根据名称匹配原则进行的数据绑定。当请求中的参数名与方法参数名称一致的时候，相应的参数值将被绑定到对应的方法参数上。比如，我们通过如下链接发起请求：

```
<a href="<c:url value='pOne.anno?age=30&author=Darren'/>">..</a>
```

为了能处理这一请求，我们就可以定义如下的处理方法：

```
@RequestMapping("/pOne.anno")
public String processMethodOne(int age,String author)
{
    // assertEquals(30,age);
    // assertEquals("Darren",author);
    return "anno/pOne";
}
```

当在处理方法中实现相应处理逻辑的时候，框架类可以保证请求参数已经完成类型转换，并绑定到该方法的对应参数上。这实际上是一个契约关系，只要保证请求参数与方法参数名称的一致，数据绑定就能够顺利完成。但是，如果把int age方法参数改为int ageOfAuthor，绑定就会出现问题。因为框架类无法在请求中找到一个名字为ageOfAuthor的请求参数。这时，int ageOfAuthor会被绑定一个null值，而int以及其他原始类型（primitive）是无法接受null值的，所以在转型期间就会抛出org.springframework.beans.TypeMismatchException。如果参数类型是String或者其他引用类型，那么绑定完成后纯粹就是一个null值了。

除了可以为基于注解的Controller的处理方法定义多个简单的参数类型，我们同样可以指定JavaBean对象引用，如代码清单26-8所示。

代码清单26-8　绑定JavaBean到请求处理方法的代码示例

```
@RequestMapping("/pOne.anno")
public String processMethodOne(int age,String author,Book book)
{
    ...
    return "anno/pOne";
}

public class Book
{
    private String bookName;
    private String bookPublisher;

    // getter和setter方法
}
```

只要同样保证请求参数名称能够与指定的JavaBean对应的属性名称匹配，数据绑定就能正确完成。假设我们通过如下链接发起一个Web请求：

```
<a href="<c:url value='pOne.anno?age=30&author=Darren&bookName=Unveil ➥
Spring&bookPublisher=unknown'/>">..</a>
```

当在处理方法中访问Book对象的时候，其属性值已经正确绑定了对应相同名称的请求参数值。同样的道理，如果像SimpleFormController那样需要一个Command对象，直接在基于注解的Controller的处理方法上，声明一个要使用的Command对象类型的方法参数即可。可以说是既方便又快捷。

2. 使用@RequestParam明确地指定绑定关系

默认的绑定行为需要我们严格遵守命名一致性原则。如果我们对此不满，想自定义绑定关系，可以求助于@RequestParam。

@RequestParam只可以标注于方法参数之上。在为我们的请求处理方法的某个方法参数标注相应的@RequestParam之后，框架类可以根据@RequestParam的信息，重新考虑要将哪个请求参数绑定到当前方法参数上。假设我们的方法定义不想使用age作为参数名，而要使用ageOfAuthor，为了保证名称为age的请求参数可以绑定到新的名称为ageOfAuthor的方法参数上，我们就可以使用@RequestParam对这一参数进行标注：

```
@RequestMapping("/pOne.anno")
public String processMethodOne(@RequestParam("age") int ageOfAuthor,String author,Book ➥
book)
{
    ...
}
```

现在，如果我们同样以http:// .../app/pOne.anno?age=30的形式发送Web请求，那么age请求参数将按照@RequestParam的指示绑定到ageOfAuthor方法参数上。

默认情况下，如果@RequestParam所指定的请求参数不存在，绑定过程中将抛出异常。这是由@RequestParam的required属性决定的，它的默认值为true，要求指定的请求参数必须存在。如果我们允许@RequestParam指定的请求参数可以时不时地外出休假（在当前请求中可以存在，也可以不存在），那么就可以将required的默认值改为false，比如：

```
@RequestMapping("/pOne.anno")
public String processMethodOne(@RequestParam(value="age",required=false) int ➥
ageOfAuthor,..)
{
    ...
}
```

如果当前请求中找不到名称为age的请求参数，那么对应的方法参数将获得相应数据类型的默认值。

3. 添加自定义数据绑定规则

Spring框架提供的数据绑定采用JavaBean规范的PropertyEditor机制进行数据转换。大多数情况下，默认注册的PropertyEditor实现类就能够满足常见数据类型的转换需求。但特殊类型的转换情况依然存在，这时我们就需要为框架提供针对特殊数据类型转换的自定义PropertyEditor实现。在基于注解的Controller中，我们可以通过如下两种方式达到这一目的。

- *使用@InitBinder标注的初始化方法*

在SimpleFormController或者其兄弟类中，如果我们想要对它们所使用的DataBinder做进

一步地定制，通常的做法是覆写父类的initBinder(..)方法。在基于注解的Controller中，做法与之相似。我们只需要在Controller实现类中声明任意一个方法，然后使用@InitBinder对该方法进行标注。这样，框架类在数据绑定之前，将保证该标注了@InitBinder的初始化方法被调用。如果我们的ReportSettingAnnotationController需要某个初始化方法来定制DataBinder（实际上我们暂时不需要），那么就可以在该类中声明一个与代码清单26-9类似的初始化方法。

代码清单26-9 使用@InitBinder声明初始化方法的代码示例

```
@Controller
@RequestMapping("/reportSetting.anno")
public class ReportSettingAnnotationController
{
    @InitBinder
    public void customizeDataBinder(WebDataBinder dataBinder)
    {
        PropertyEditor propEditor = ..;
        dataBinder.registerCustomEditor(SomeDataType.class,propEditor);
        ...
    }
    @RequestMapping(method=RequestMethod.GET)
    public String displayReportSettings(..)
    {
        ...
    }
    @RequestMapping(method=RequestMethod.POST)
    public String updateReportSetting(..)
    {
        ...
    }
}
```

26

初始化方法不能有返回值，而且至少应该有一个类型为org.springframework.web.bind.WebDataBinder的方法参数，否则，是要对哪个DataBinder进行定制呢？声明WebDataBinder类型的方法参数，使得框架可以将它所使用的DataBinder引用传给我们，以做进一步的定制操作。

初始化方法的方法参数除了不能使用Command对象以及相关的Errors/BindingResult对象之外，一个典型的基于注解的Controller的处理方法可以使用的方法参数类型都可以使用。不过，大多情况下，在WebDataBinder之后提供一个WebRequest或者Locale对象就足够了。这允许我们根据情况对DataBinder做相应定制。

- 指定自定义的WebBindingInitializer

使用@InitBinder标注的初始化方法只能对一个Controller对应的DataBinder做定制。如果某些Controller可以共享相同的一段定制逻辑，比如，几个Controller实现类都用到了某一特殊数据类型的绑定，那么它们就可能使用同一PropertyEditor实例。这时，为AnnotationMethodHandler-Adapter指定一个自定义的org.springframework.web.bind.support.WebBindingInitializer实例，可以避免在每个Controller中都重复定义几乎相同逻辑的@InitBinder初始化方法。

AnnotationMethodHandlerAdapter是Spring MVC框架为基于注解的Controller提供的HandlerAdaptor实现类。如果没有在DispatcherServlet的WebApplicationContext中明确指定的话，DispatcherServlet将默认初始化一个该类的实例来使用。现在我们要为Annotation-

MethodHandlerAdapter提供自定义的WebBindingInitializer实现，所以，不得不在DispatcherServlet的WebApplicationContext中明确指定一个该类对应的bean定义了。

如果我们拥有如代码清单26-10所示的WebBindingInitializer实现类。

代码清单26-10 自定义WebBindingInitializer代码示例

```
public class GenericBindingInitializer implements WebBindingInitializer {

    public void initBinder(WebDataBinder binder, WebRequest request) {
        PropertyEditor propEditorOne = ..;
        binder.registerCustomEditor(SomeDataType.class, propEditorOne);

        //如果需要，可以注册更多propertyEditor
        ...
    }
}
```

为了让它可以为多个基于注解的Controller的数据绑定提供服务，我们在DispatcherServlet的WebApplicationContext中添加如下配置内容：

```
<bean class="org.springframework.web.servlet.mvc.Annotation.AnnotationMethodHandlerAdapter">
    <property name="webBindingInitializer">
        <bean class="..GenericBindingInitializer" />
    </property>
</bean>
```

DispatcherServlet现在可以根据类型获取这一AnnotationMethodHandlerAdapter来处理基于注解的Controller。

26.3.3 使用@ModelAttribute 访问模型数据

在介绍@ModelAttribute之前，我们要先完成ReportSettingAnnotationController的一部分功能，已有的功能介绍已经可以完成80%的工作了。

在介绍@RequestMapping的时候，我们已经就ReportSettingAnnotationController所要处理的请求方法做了个骨架性的定义，现在要进一步完善它们。ReportSettingAnnotationController的displayReportSettings(..)方法负责GET方法请求的处理，也就是负责显示现有的报表设定情况。在该方法中，我们需要调用相应的服务来获取现有的报表设定信息，然后添加到模型数据中返回给视图进行显示。为了能够访问模型数据，我们可以声明一个ModelMap类型的方法参数。作用就不用多说了吧？见代码清单26-11。

代码清单26-11 初步补充血肉后的ReportSettingAnnotationController定义

```
@Controller
@RequestMapping("/reportSetting.anno")
public class ReportSettingAnnotationController
{
    public static final String FORM_VIEW_NAME = "anno/reportSetting";
    @Autowired
    private IReportSettingManager reportSettingManager;

    @RequestMapping(method=RequestMethod.GET)
    public String displayReportSettings(ModelMap model)
    {
        ReportSettings reportSettings = getReportSettingManager().getReportSettings();
```

```
        model.addAttribute("command",reportSettings);
        return FORM_VIEW_NAME;
    }
    @RequestMapping(method=RequestMethod.POST)
    public String updateReportSetting(..)
    {
        ...
    }
    // 对应reportSettingManager的setter和getter方法定义
}
```

使用Spring 2.5的@Autowired为ReportSettingAnnotationController提供IReportSettingManager依赖的注入，这样可以避免通过XML配置来完成依赖注入关系的确定。

要为视图渲染提供更多模型数据，除了像我们现在这样，通过ModelMap方法参数直接添加之外，基于注解的Controller还为我们提供了另一种选择，那就是使用@ModelAttribute。@ModelAttribute可以应用在方法级别和方法参数上。如果将@ModelAttribute标注在某个方法上，该方法所返回的数据将被添加到模型数据中。这类似于SimpleFormController的referenceData()方法完成的工作。比如，我们可以将displayReportSettings()方法中添加模型数据的逻辑独立出来，新的代码如代码清单26-12所示。

代码清单26-12 使用@ModelAttribute的代码示例

```
@Controller
@RequestMapping("/reportSetting.anno")
public class ReportSettingAnnotationController
{
    public static final String FORM_VIEW_NAME = "anno/reportSetting";
    @Autowired
    private IReportSettingManager reportSettingManager;

    @ModelAttribute("command")
    public ReportSettings referenceDataLikeMethod()
    {
        ReportSettings reportSettings = getReportSettingManager().getReportSettings();
        return reportSettings;
    }

    @RequestMapping(method=RequestMethod.GET)
    public String displayReportSettings(ModelMap model)
    {
        return FORM_VIEW_NAME;
    }
    @RequestMapping(method=RequestMethod.POST)
    public String updateReportSetting(..)
    {
        ...
    }
    // 对应reportSettingManager的getter和setter方法
}
```

referenceDataLikeMethod()方法所返回的数据将以@ModelAttribute("command")所指定的"command"作为键值添加到模型数据中。

标注在方法级别上的@ModelAttribute只提供了对模型数据的“存”操作。如果将@ModelAttribute标注到处理方法的方法参数上的话，也就是应用在方法参数级别，那么“存取操作”就算

26

圆满了。即将要实现的updateReportSetting(..)方法展示了@ModelAttribute的这一功能（见代码清单26-13）。

代码清单26-13 标注于方法参数的@ModelAttribute使用代码示例

```
@Controller
@RequestMapping("/reportSetting.anno")
public class ReportSettingAnnotationController
{
    public static final String FORM_VIEW_NAME = "anno/reportSetting";
    @Autowired
    private IReportSettingManager reportSettingManager;

    @RequestMapping(method=RequestMethod.GET)
    public String displayReportSettings(ModelMap model)
    {
        ReportSettings reportSettings = getReportSettingManager().getReportSettings();
        model.addAttribute("command",reportSettings);
        return FORM_VIEW_NAME;
    }
    @RequestMapping(method=RequestMethod.POST)
    public String updateReportSetting(@ModelAttribute("command") ReportSettings ➥
    reportSettings, BindingResult result, SessionStatus status)
    {
        if(result.hasErrors())
        {
            return FORM_VIEW_NAME;
        }
        getReportSettingManager().updateReportSettings(reportSettings);
        status.setComplete();
        return "redirect:reportSetting.anno";
    }
    // 对应reportSettingManager的getter和setter方法定义
}
```

框架类将保证在调用updateReportSetting(..)方法处理请求提交的时候，从模型数据中获得以@ModelAttribute("command") 标注的 ReportSettings reportSettings。ReportSettings reportSettings在该方法定义中同时身兼Command对象的重任，所以，我们在该参数之后同时添加了BindingResult result作为第二个方法参数。通过BindingResult result，我们可以获得数据绑定结果，然后根据结果来决定后继流程该如何处理。那么，SessionStatus在这里是干什么的呢？

如果我们现在使用以上的ReportSettingAnnotationController处理表单提交，就会发现updateReportSetting(..)方法不能“正确”的工作。为什么呢？我们在GET方法请求中放到模型数据中的Command对象，与POST方法提交的请求中从模型数据中获取的Command对象是不同的，因为这是两个独立的请求。为了保证GET请求和POST请求使用模型数据中同一个Command对象，我们需要将GET请求中的Command对象存入Session中。这样，POST形式的请求就可以从Session中获取同一个Command对象了。但是，如何在基于注解的Controller中做到这一点呢？答案就是使用@SessionAttribute。

26.3.4 通过@SessionAttribute 管理 Session 数据

为了能够访问Session，我们当然可以为每个处理方法声明一个HttpSession类型的方法参数。不过，这看起来比较“土”，更多时候，我们可以借助@SessionAttribute来标注哪些数据需要通过session

进行管理。

@SessionAttribute只应用在类型声明上，当我们将ReportSettingAnnotationController标注如代码清单26-14所示样子的时候。

代码清单26-14 @SessionAttribute使用代码示例

```
@Controller
@RequestMapping("/reportSetting.anno")
@SessionAttributes("command")
public class ReportSettingAnnotationController
{
    public static final String FORM_VIEW_NAME = "anno/reportSetting";
    @Autowired
    private IReportSettingManager reportSettingManager;

    @RequestMapping(method=RequestMethod.GET)
    public String displayReportSettings(ModelMap model)
    {
        ReportSettings reportSettings = getReportSettingManager().getReportSettings();
        model.addAttribute("command",reportSettings);
        return FORM_VIEW_NAME;
    }
    @RequestMapping(method=RequestMethod.POST)
    public String updateReportSetting(@ModelAttribute("command") ReportSettings ➥
    reportSettings, BindingResult result, SessionStatus status)
    {
        if(result.hasErrors())
        {
            return FORM_VIEW_NAME;
        }
        getReportSettingManager().updateReportSettings(reportSettings);
        status.setComplete();
        return "redirect:reportSetting.anno";
    }
    // 对应reportSettingManager的getter和setter方法定义
}
```

26

在GET请求中添加到模型数据的Command对象将被存入Session中，而在POST请求处理期间从模型数据获得的，就是当初存入Session的同一个Command对象了。SessionStatus的作用是将使用完的数据清除出Session。

@SessionAttributes允许以属性名称或者类型两种方法，来表明将哪些数据通过Session进行管理。以上我们使用的是指定属性名称的方式，但通过类型来指定也是可行的，如下所示：

```
@SessionAttributes(types=ReportSettings.class)
```

两种指定方式可以在开发过程中根据情况选用。

到此为止，我们接触了基于注解的Controller所有的特性。如果我们使用代码清单26-15给出的视图模板结合最终完成的ReportSettingAnnotationController，那么，最初设定的功能需求即告完成。

代码清单26-15 ReportSettingAnnotationController对应的视图模板代码示例

```
<form:form action="reportSetting.anno">
    <form:errors path="*"></form:errors><br/>
    <display:table name="command.reportSettings" id="row">
```

```
        <display:caption>报表设定</display:caption>
        <display:column property="reportName" title="报表名称"></display:column>
        <display:column title="保存期限">
            <form:radiobutton path="reportSettings[${row_rowNum-1}].holdingYears" ➥
            value="5" label="五年"/>
            <form:radiobutton path="reportSettings[${row_rowNum-1}].holdingYears" ➥
            value="10" label="十年"/>
        </display:column>
    </display:table>
    <input type="submit" value="更新"/>
</form:form>
```

最后，希望你能在使用基于注解的Controller过程中融入更多的轻松与快乐！

26.4 小结

Spring 2.5中新添加的基于注解的Controller形式，可以让我们以更加灵活高效的方式开发基于Spring MVC的Web应用。实际上，这种基于注解的Controller形式并没太多新意，只是在现有Spring MVC框架的基础上做出的自定义功能扩展。本章在分析基于注解的Controller形式的实现原理的基础上，对基于注解的Controller形式给出了详细的介绍，并进行了深入地剖析。相信你在完成本章内容之后，已经对基于注解的Controller形式爱不释手了吧！

第27章 Spring MVC之扩展篇

本章内容

- Spring MVC也Convention Over Configuration
- Spring 3.0展望

27.1 Spring MVC 也 Convention Over Configuration

随着Ruby On Rails这一Web开发框架的流行，Convention Over Configuration的理念也越来越受到大家的欢迎。而Spring MVC的良好设计，也使得扩展它并加入Convention Over Configuration的元素变得并非什么难事。

> 注意　Spring MVC提供的Convention Over Configuration支持，是在Spring 2.0发布之后引入的。在Spring 2.5发布之后，我们更倾向于基于注解的Controller。在扩展篇依然对Spring MVC中Convention over Configuration进行介绍的目的是，帮助大家扩展思路，让大家可以进一步地了解Spring MVC的良好设计所赋予我们的扩展现有框架的能力。

27.1.1 Convention Over Configuration 简介

Convention Over Configuration，国内技术社区通常称其为“惯例优于配置”或者“惯例优先原则”或者其他称呼。它所倡导的理念是，如果系统中某些映射关系可以通过事先约定的规则确定，那么优先使用事先约定的规则来确定当前映射关系，这样可以避免将所有的映射关系通过配置的方式来表述，进而大大减少配置量，提高开发效率。在Java平台的ORM产品中，PO（Persistence Object）与数据库表之间的映射关系，之前大多是通过配置文件来表述的，而Ruby On Rails所使用的ActiveRecord这一ORM解决方案则按照约定规则，直接根据对象类名推测该类所映射到的数据库表。比如，User类将映射到users数据库表，也就是说，对象与数据库表之间的命名约定确定了二者之间的映射关系，而不需要任何附加的配置文件来明确这一点。相对于Java平台ORM产品早先的做法，后者将大大减少配置的工作量。Java平台最初更多偏重于使用外部配置，是因为外部配置可以带给应用程序和开发人员很大的灵活性。但随着系统规模的增长，配置的工作量显然也是有增无减，通过Convention Over Configuration的方式，虽然失去了部分的灵活性，但可以极大减少配置的工作量，提高日常开发效率。权衡利弊之后，我们就可以做出自己的选择了。

实际上，Convention Over Configuration的理念与使用外部配置并不冲突。使用Convention Over Configuration并不意味着我们就得丢弃外部配置方式，我们依然可以通过外部配置来覆盖默认约定所

确定的映射关系。甚至，在通过事先的约定无法确定所需要的映射关系的情况下，外部配置将成为“后备军”以确保万无一失。所以，在Convention Over Configuration中，Convention与Configuration的关系更多是协作关系，只是，Convention会优先上场罢啦！从这一角度来说，将Convention Over Configuration称为“约定优先于配置”或许更贴切一些。不过，为了避免翻译造成的混乱，余下部分我们将一直使用原生的Convention Over Configuration来表述这一概念。

27.1.2　Spring MVC 中的 Convention Over Configuration

Spring从来不会将好的理念和实践方式拒之门外，所以，Spring框架发布2.0版本的同时，为Spring MVC添加了Convention Over Configuration的支持，并借此进一步简化了Spring MVC Web框架的使用。

为了演示Spring MVC如何支持Convention Over Configuration，我们先对Web请求处理过程中所涉及的角色之间的映射关系做事先约定：

- 客户端以http://server:port/application/sample.do的形式发起Web请求；
- 服务器端将根据请求的URL，选用类名为`SampleController`的Handler来处理当前请求；
- 请求处理完成后，同样根据请求的URL，选择名称为sample.jsp的模板文件进行视图渲染；

下面让我们看看，在Spring MVC中如何实现这一约定下的请求处理过程。

1. Web请求与Handler之间的约定

在Spring MVC中，Web请求与请求处理Handler之间的关系是由`HandlerMapping`来管理的。不过，通常情况下使用的`BeanNameUrlHandlerMapping`或者`SimpleUrlHandlerMapping`都是通过配置来管理请求与Handler之间的映射关系，要使用Convention Over Configuration显然不能依靠它们。为了让`DispatcherServlet`能够根据约定，使用请求URL所“极力呼吁”的Controller实现类，我们需要提供`BeanNameUrlHandlerMapping`或者`SimpleUrlHandlerMapping`之外的`HandlerMapping`实现。该自定义`HandlerMapping`实现类将根据请求的URL，到容器中选取约定名称或者约定类型的Controller实现类来进行当前Web请求的处理。要真去实现这么一个自定义`HandlerMapping`实现类也并非难事，不过好消息是，2.0版本之后的Spring MVC已经提供了`org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping`来完成同一历史使命。

`ControllerClassNameHandlerMapping`在处理Controller与具体Web请求之间的映射关系的时候，使用的约定规则，如下所述。

- 对于简单的Controller实现来说，`ControllerClassNameHandlerMapping`将去除了“Controller”后缀的类名作为映射的请求路径，比如：
  - CoCController -> /coc*
  - SpringController -> /spring*

  默认情况下，转换后的结果均为小写，但我们可以通过将`ControllerClassNameHandler-Mapping`的caseSensitive设置为true更改这一默认行为。
- 对于`MultiActionController`来说，生成映射的规则略有改变，采用了类似Servlet映射中的前缀映射规则：
  - CoCController -> /coc/*
  - SpringController -> /spring/*

    结合`InternalPathMethodNameResolver`使用的话，恰好可以达到与Ruby On Rails通常做法类似的映射效果。

如果我们对默认的约定规则还不满意，通过设置或者更改`ControllerClassNameHandler-`

Mapping的相应配置项可以进一步对约定规则进行定制，如下所述。

pathPrefix。该属性允许在默认约定规则的基础上，让相应的Controller去处理带有指定路径前缀的请求，比如，在如下ControllerClassNameHandlerMapping基础上：

```
<bean  id="cocHandlerMapping" ➥
class="org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping"
p:order="1" p:pathPrefix="module"/>
```

SampleController处理的请求变成对应http://server:port/application/module/sample.do的形式。

basePackage。假设我们的Controller实现类按照功能模块划分，定义在不同的包下：

- cn.spring21.simplefx.controller.**group**.SampleController
- cn.spring21.simplefx.controller.**rate**.SampleController
- cn.spring21.simplefx.controller.**report**.SampleController

那么，在将ControllerClassNameHandlerMapping的basePackage属性设定为cn.spring21.simplefx.controller.的时候，对应三种不同源代码包下的Controller的请求路径就变成：

- cn.spring21.simplefx.controller.group.SampleController -> group/sample.do
- cn.spring21.simplefx.controller.rate.SampleController -> rate/sample.do
- cn.spring21.simplefx.controller.report.SampleController -> report/sample.do

这对于按照功能模块进行管理的实践比较实用。

其他可以使用的定制属性还有excludedClasses和excludedPackages，允许我们将某些Controller实现类排除在ControllerClassNameHandlerMapping所管理的映射之外。如果需要，可以参阅ControllerClassNameHandlerMapping的Javadoc文档以获取更多信息。

如果没有为DispatcherServlet明确声明一个可用的HandlerMapping实例，那么DispatcherServlet将在初始化的时候默认启用一个ControllerClassNameHandlerMapping（随同的还有BeanNameUrlHandlerMapping）。除非要对ControllerClassNameHandlerMapping的约定规则进行定制，通常情况下，不需要在DispatcherServlet的WebApplicationContext中声明它，就能使用Convention Over Configuration的支持了。现在，如果要为http://server:port/application/sample.do形式的请求提供Controller实现的话，只需要按照以上约定，定义一个SampleController，然后注册到IoC容器中即可，如下所示： 27

```
<bean id="sample" class="..SampleController"/>
```

ControllerClassNameHandlerMapping将可保证Web请求到Controller的正确映射。如果是使用BeanNameUrlHandlerMapping或者SimpleUrlHandlerMapping，除了添加Controller的bean定义之外，还得添加映射信息的配置。现在，这些都可以免了。

> **注意** 最初在注册相应Controller实例的时候，可能需要必须指定符合约定的bean定义名称，比如id= “sample”。不过，最新的Spring版本中，即使不声明bean定义名称也是可以的。

2. **ModelAndView中的约定**

在相应的视图模板访问模型数据的时候，需要根据数据项的键（key）来获取对应的数据值。所以，通常情况下，我们都是按照如下形式将模型数据添加到ModelAndView的：

```
ModelAndView mav = new ModelAndView();
mav.addObject("key1", value1);
mav.addObject("key2", value2);
...
```

但现在引入Convention Over Configuration理念之后，即使我们不指定添加到ModelAndView中的数据所对应的键，按照约定，ModelAndView依然可以为添加的数据提供一个默认的键（key）来标志它，例如：

```
ModelAndView mav = new ModelAndView();
Customer cust = ...;
mav.addObject(cust);          // -> key=customer

Customer[] customers = ...;
mav.addObject(customers);  // -> key=customerList
...
```

基本上，如果添加的数据类型为单一对象的话，数据对应的类名转换为小写，即为其所对应的键。而如果添加的数据类型为集合或者数组的话，集合或者数字中的对象类型所对应的类名转换为小写然后添加List后缀所得的结果，将作为其在模型中的键（key）。更多信息可以参考Spring Reference文档，此处不再赘述。

3. Web请求与视图之间的约定

相应Controller处理完当前Web请求之后，会通过ModelAndView返回后继流程要使用的逻辑视图名。但通常情况下，我们都是明确指定ModelAndView所返回的逻辑视图名是什么。对于Convention Over Configuration来说，这显然是多此一举的。所以，要朝着Convention Over Configuration迈进的话，我们就不应该每次去明确指定当前请求对应的逻辑视图名是什么。可是，这样的话，ViewResolver如何决定返回哪个视图呢？

坦率地讲，不能从ModelAndView中取得一个逻辑视图名的话，ViewResolver确实无能为力，它不能凭空造一个视图出来，或者随便挑一个返回，而且，ViewResolver接口定义中并不涉及HttpServletRequest，它显然也无法根据请求的URL来推测可能使用的逻辑视图名是哪个。为了能够让ViewResolver从请求URL中根据约定提取默认的逻辑视图名，Spring MVC引入了org.springframework.web.servlet.RequestToViewNameTranslator。当框架内部不能从ModelAndView中获取可用的逻辑视图名的时候，由RequestToViewNameTranslator根据约定从当前请求的URL中提取。显然，该接口必定与HttpServletRequest有所关联，如下所示：

```
public interface RequestToViewNameTranslator {
   String getViewName(HttpServletRequest request) throws Exception;
}
```

在RequestToViewNameTranslator按照约定从请求的URL中提取到可用的逻辑视图名之后，ViewResolver就可以正常工作了。

RequestToViewNameTranslator接口在Spring MVC框架内只有一个实现，即org.springframework.web.servlet.view.DefaultRequestToViewNameTranslator。DispatcherServlet在初始化的时候，将默认使用该实现类来处理Convention Over Configuration情况下，请求URL到视图的映射关系。DefaultRequestToViewNameTranslator在处理请求到视图名的映射关系的时候，行为上与ControllerClassNameHandlerMapping处理请求到Handler的映射关系是类似的，如：

- http://.../application/sample.do -> sample
- http://.../application/module/sample.do -> module/sample

另外，我们同样可以通过DefaultRequestToViewNameTranslator的相应属性，对默认的映射约定进行定制，比如通过prefix或者suffix属性为默认提取的视图名添加相应的前缀或者后缀等。

当ViewResolver根据RequestToViewNameTranslator从请求URL中提取的逻辑视图名返回对

应的视图的时候，整个按照Convention Over Configuration方式处理Web请求的过程就算完成了！

27.2 Spring 3.0 展望

在Spring 3.0中，针对Spring MVC的更新幅度应该算是最大的了吧！

首先，在Spring 2.5发布之后，“基于注解的Controller”就已经成为最被推荐的Spring MVC实践方式。在Spring 3.0中，这种实践倡导将被进一步增强，也就是说，Spring MVC将更倾向于成为Spring @MVC，以注解驱动的Spring MVC开发将在Spring 3.0中红遍整个天。

其次，Spring团队在发布了Spring 3.0 M2之后的博客中阐述到，他们还可能考虑在Spring 3.0中追加对JSF 2.0的正式支持，但这依然没有成为定局，能不能成为现实，还要看增加这样的支持是否会对现有的Spring MVC框架造成很大的变更，或者也有可能针对JSF 2.0变更现有的Spring MVC实现架构。所有这些都只能在Spring 3.0发布之后，才可知晓Spring团队是如何做出决定的。

在Spring 3.0中，针对REST的支持将成为重头戏，按照Spring团队博客上的相关信息可以知道（可以参看http://blog.springsource.com/2009/03/08/rest-in-spring-3-mvc/），针对REST的支持将被集成到Spring MVC的核心当中，这将大大简化使用Spring MVC或者说Spring @MVC开发REST应用的难度。

根据已知的信息可以知道，Spring 3.0将为Spring MVC更新或者说增加如下装备。

- ContentNegotiatingViewResolver。为了能够适应REST对视图类型显示的需要，自定义的ViewResolver被引入；
- URI Template。结合@PathVarible，我们可以定义满足URI Template形式的请求处理映射，比如：

```
// 以下代码摘自SpringSource Blog
@RequestMapping(value="/hotels/{hotel}/bookings/{booking}", method=RequestMethod.GET)
public String getBooking(@PathVariable("hotel") long hotelId, @PathVariable("booking")➥
long bookingId, Model model) {
    Hotel hotel = hotelService.getHotel(hotelId);
    Booking booking = hotel.getBooking(bookingId);
    model.addAttribute("booking", booking);
    return "booking";
}
```

  通过@PathVarible标注的方法参数将被绑定请求映射中相应位置的值，比如，针对以上代码定义，在/hotels/1/bookings/2的情况下，1将被绑定到hotelId方法参数，而2将被绑定到bookingId方法参数。

- 各种新的View实现。这包括AbstractAtomFeedView、AbstractRssFeedView、MarshallingView以及JacksonJsonView。
- HiddenHttpMethodFilter。虽然HTTP定义有PUT、POST、GET和DELETE四种方法，但通常HTML只支持GET和POST两种。要实现REST形式的服务，我们就得“曲线救国”，比如通过JavaScript来发送相应请求。HiddenHttpMethodFilter是一个Servlet的Filter实现，其作用在于，它可以拦截并转换相应的请求方法，使得这一切对后端透明。Spring 3.0中通过增强的form标签库和HiddenHttpMethodFilter为基于REST的应用开发提供了一套完备的服务。

另外，Spring 3.0也将对Spring Portal MVC进行升级，以支持Portlet 2.0 API。与Spring MVC将更倾向于Spring @MVC相类似，Spring Portal MVC注定也将向这个方向发展，相应的注解将被添加或者被强化。

总之，让我们拭目以待，期待Spring 3.0为我们带来更多的惊喜！

27.3 小结

显然，Spring MVC良好的架构设计使得引入Convention Over Configuration的理念变得极其轻松，只提供一个自定义的`HandlerMapping`和扩展的`RequestToViewNameTranslator`就搞定了。实际上，纵观Spring MVC，我们会发现，无论是Spring 2.5后引入基于注解的Controller，还是Spring 2.0之后引入的Convention Over Configuration，它们都是在保持Spring MVC框架总体一致性的基础上添加的自定义扩展而已。如果需要，我们同样可以通过扩展Spring MVC来引入更多新的特性或者理念，不是吗？

Part 7 第七部分

Spring框架对J2EE服务的集成和支持

Java EE（原来的J2EE）平台由许多企业级应用服务组成，比如JMS、JTA、JavaMail等。这些服务规范所给出的API设计虽然强大，但往往过于底层。如果不能对这些API的使用做适度的封装，开发人员在日常开发过程中势必消耗过多的精力在API的调试或者重复编码上。鉴于此，Spring框架通过近乎一致的方式对各种Java EE服务API的使用进行了合适的封装，以此来帮助我们简化日常的开发工作。这正是本部分将要展示的内容。

本部分内容

- 第28章　Spring框架内的JNDI支持
- 第29章　Spring框架对JMS的集成
- 第30章　使用Spring发送E-mail
- 第31章　Spring中的任务调度和线程池支持
- 第32章　Spring框架对J2EE服务的集成之扩展篇
- 第33章　Spring远程方案

第 28 章

Spring框架内的JNDI支持

本章内容

- JNDI简单回顾
- Spring框架内JNDI访问的基石——JndiTemplate
- JNDI对象的依赖注入——JndiObjectFactoryBean

28.1 JNDI 简单回顾

JNDI（Java Naming and Directory Interface，Java命名与目录接口），其主要目的是为了统一各种命名与目录服务（naming service/directory service）的访问接口，这与JDBC规范的提出目的有异曲同工之妙。整个JNDI的架构由API和SPI（Service Provider Interface）两部分组成，如图28-1所示。

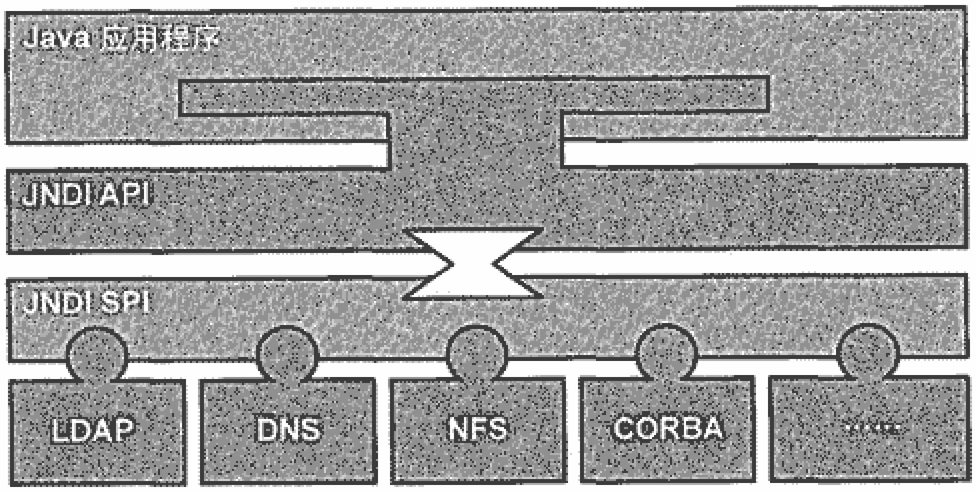

图28-1 JNDI架构示意图

JNDI API主要公开给Java应用程序使用，它为Java应用程序访问各种命名与目录服务提供了统一的接口。不管Java应用程序将访问的命名和目录服务如何变换，只要Java应用程序是通过JNDI API进行命名和目录服务访问的，就可以"无视"各种具体命名和目录服务的差异和变更。JNDI SPI主要公开给具体命名或者目录服务提供商（Vendor）使用，它为各种具体的命名和目录服务产品提供了一个扩展层。提供商如果愿意加入JNDI规范的行列，就可以根据JNDI SPI规范，给出针对自己命名或者目录服务产品的SPI实现，然后集成到JNDI架构中。除非你被委以重任，要为某种命名和目录服务产品提供JNDI的SPI实现，大部分情况下，我们都只是直接跟JNDI的API打交道。

注意 JNDI的架构设计拥有良好的通用性（从API角度）和扩展性（从SPI角度），这使得我们可以以统一的方式访问任何命名与目录服务，只要有相应的SPI实现就行。不过，或许你也注意到了，大多数时候，我们是直接使用Java应用服务器所提供的JNDI服务实现（各个应用服务器需要根据Java EE规范提供相应的JNDI服务），但不要让这种特例蒙蔽了我们的双眼，JNDI实际上很强大的哦！

在Java EE平台上，JNDI更多的是为资源的访问和部署提供一个“隔离层”。如果我们通过JNDI进行相应资源的部署，并通过JNDI对这些绑定到JNDI服务的资源进行访问，那么，对资源进行访问的应用程序以及对具体资源的部署就都能够独立演化，即使系统管理员因为某些原因要对某些资源的部署做相应的调整，使用JNDI对这些资源进行访问的应用程序也不会因此受影响。

现有的许多Java EE服务都依赖于JNDI来访问相应资源，比如JMS需要通过JNDI获取`ConnectionFactory`，EJB需要通过JNDI获取HOME接口的引用等，为了使大家能够更好的理解Spring框架对这些JavaEE服务的集成，出于铺垫的考虑，我们不得不把了解“Spring对JNDI的支持”作为我们本部分探索之旅的始发站，现在要发车啦！

注意 更多有关JNDI的信息可以参考Sun公司提供的“The JNDI Tutorial”（http://java.sun.com/products/jndi/tutorial/）或者网上的相关资源。

28.2 Spring 框架内 JNDI 访问的基石——JndiTemplate

实际上，相对于JDBC API来说，JNDI API的使用看起来要简单得多。不过，与我们所要达到的要求还有一定的距离，代码清单28-1中是通常情况下的JNDI API的使用示例。

代码清单28-1 JNDI API的使用示例

```
public DataSource getDataSourceInTraditionalStyle() throws NamingException
{
    Hashtable env = new Hashtable();
    env.put(Context.INITIAL_CONTEXT_FACTORY, "..");
    env.put(Context.PROVIDER_URL, "..");
    // 其他可能的设置……
    Context context = null;
    try
    {
        context = new InitialContext(env);
        return (DataSource)context.lookup("jdbc/yourDataSource");
    }
    finally
    {
        if(context != null)
        {
            try {
                context.close();
            } catch (NamingException e) {
                e.printStackTrace(); // 不要这样做。
            }
        }
    }
}
```

28

实际上，在代码清单28-1中，我们只需要一个DataSource。我想其中只有(DataSource) context.lookup("jdbc/yourDataSource")对我们是紧要的。要是每次通过JNDI来获取相应的资源引用，都书写这么一堆逻辑近乎不变的代码，肯定是怎么看怎么不爽啦！

你可以说这是一个传统，你也可以说这是Spring框架在API设计上的一致性，模板方法模式在这里又一次与我们相遇了。Spring框架提供了org.springframework.jndi.JndiTemplate这一Helper类以帮助我们简化一系列的JNDI操作，包括对象的查找、绑定或者解除绑定等。有了之前一系列模板方法模式使用上的"洗礼"之后，JndiTemplate看起来也容易理解得多。

JndiTemplate的核心方法定义如代码清单28-2所示。

代码清单28-2　JndiTemplate核心模板方法定义代码摘录

```
public Object execute(JndiCallback contextCallback) throws NamingException {
    Context ctx = createInitialContext();
    try {
        return contextCallback.doInContext(ctx);
    }
    finally {
            try {
                ctx.close();
            }
            catch (NamingException ex) {
                logger.debug("Could not close JNDI InitialContext", ex);
        }
    }
}
```

JndiCallback回调接口负责提供具体的JNDI访问逻辑，而底层的JNDI基础设施管理则由当前模板方法来负责。在JNDI访问逻辑需要进一步定制的时候，我们才会考虑execute()方法加JndiCallback的使用形式。对于常见的JNDI访问操作，JndiTemplate在execute()核心模板方法上构建的各种便利性模板方法将是首选。下面的代码片段演示了使用JndiTemplate进行JNDI资源访问与传统API使用模式上的差别：

```
public DataSource getDataSourceViaJndiTemplate() throws NamingException{
    Properties env = new Properties();
    env.setProperty(Context.INITIAL_CONTEXT_FACTORY, "..");
    env.setProperty(Context.PROVIDER_URL, "..");
    // 其他可能的设置……
    JndiTemplate jndiTemplate = new JndiTemplate(env);
    return (DataSource)jndiTemplate.lookup("jdbc/yourDataSource");
}
```

当然，实际情况下我们不会把初始化JNDI的环境设置数据硬编码到代码中，将JndiTemplate添加到Spring的IoC容器然后注入给依赖的对象或许会更好，如下所示：

```
<bean id="jndiTemplate" class="org.springframework.jndi.JndiTemplate">
    <property name="environment">
        <props>
            <prop key="java.naming.factory.initial">..</prop>
            <prop key="java.naming.provider.url">..</prop>
        </props>
    </property>
</bean>
```

现在，谁需要JNDI访问支持，将JndiTemplate注入给它就行。而且，我想现在访问JNDI的代码就真的只有一行啦。

28.3 JNDI 对象的依赖注入——JndiObjectFactoryBean

尽管JndiTemplate在一定程度上简化了JNDI的访问操作，但并不意味着我们就可以不分场合地滥用JndiTemplate。那些依赖于绑定到JNDI服务某项资源的对象，实际上并不关心该项资源到底是来自JNDI服务还是其他位置。所以，如果我们在这些对象内直接使用JndiTemplate甚至是JNDI的Context为其获取相应依赖的话，势必造成当前业务对象与Spring API和JNDI API的过紧耦合。为了避免这种问题，我们应该将JNDI资源访问相关逻辑剥离出当前对象，由其他辅助对象来实现这些逻辑，并将辅助对象获取到的被依赖对象注入到需要它的对象中。Spring框架提供的org.springframework.jndi.JndiObjectFactoryBean将帮助我们完成类似的使命。

JndiObjectFactoryBean是Spring的FactoryBean实现类，负责根据请求的jndi名称，查找并返回绑定到JNDI服务的相应资源引用。帮助JndiObjectFactoryBean完成所有JNDI访问操作的，自然就是JndiTemplate。鉴于FactoryBean的特殊性质，在我们将JndiObjectFactoryBean作为依赖注入给相应对象之后，该对象获取的依赖对象将是JndiObjectFactoryBean通过JNDI所获取的资源引用，而不是JndiObjectFactoryBean本身。所以，在我们声明如下依赖DataSource的对象定义之后：

```
public class DataSourceRequired{
    private DataSource dataSource;
    ..
    // getter和setter等方法定义
}
```

如下的bean定义将保证该对象获取到来自JNDI的DataSource引用：

```
<bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean"➥
p:jndiName="jdbc/yourDataSource">
</bean>

<bean id="target" class="..DataSourceRequired" p:dataSource-ref="dataSource"/>
```

当然，Spring 2.0发布之后，以上配置内容在引入了jee命名空间的XSD配置文件中可以进一步简化：

```
<jee:jndi-lookup id="dataSource" jndi-name="jdbc/yourDataSource"/>

<bean id="target" class="..DataSourceRequired" p:dataSource-ref="dataSource"/>
```

不管最终我们将使用何种配置形式，我们的目的都是相同的，那就是通过JndiObjectFactory-Bean的存在，让对象与其所依赖的对象与JNDI不发生任何的耦合。

默认情况下，JndiObjectFactoryBean在初始化的时候就会查找指定名称的JNDI对象引用，然后缓存之。对于那些不经常变更的JNDI资源来说，这种默认行为可以避免性能上的损失。不过，如果绑定到JNDI的资源经常变动，或者我们希望开发过程中能够实时反映资源的变更情况的话，就可以通过设置JndiObjectFactoryBean的相应属性来定制其行为。表28-1是相应属性与其所控制行为之间的一个简单说明。

表28-1 JndiObjectFactoryBean行为定制属性表

属 性	默认值	行 为
lookupOnStartup	true	该属性默认值使得JndiObjectFactoryBean在实例化之初就查找并获取jndiName属性所对应的JNDI资源。如果将其设置为false，需要同时设置proxyInterfaces属性以表明JNDI资源所对应的类型。这样，JndiObjectFactoryBean将通过JndiObjectTargetSource返回请求类型的一个代理对象，从而可以在被请求的时候再执行查找
cache	true	默认值使得JndiObjectFactoryBean将缓存查找结果。如果要避免查找结果的缓存，可以将其设置为false，并同时指定proxyInterfaces，原因与lookupOnStartup的变更一样。因为JndiObjectFactoryBean背后将使用JndiObjectTargetSource构建查找的代理对象，采用基于接口的代理机制，所以，proxyInterfaces在这两种情况下是必须的
defaultObject	null	当查找失败的时候，如果defaultObject非空，JndiObjectFactory-Bean可以将其作为后备结果返回，否则将抛出异常

从表28-1中我们可以知道，当将lookupOnStartup和cache设置为false的时候，需要指定proxyInterfaces属性以使得JndiObjectTargetSource可以构建相应的代理对象。这个JndiObject-TargetSource实际上就是AOP的TargetSource实现，它的默认行为与JndiObjectFactoryBean类似，也是默认情况下在实例化的时候执行查找，然后缓存查找结果。不过同样可以通过lookupOnStartup和cache属性来变更它的这种默认行为。所以，基本上JndiObjectFactoryBean和JndiObject-TargetSource可以满足相同的查找需求，在使用的时候根据情况选择其中一个就可以了。

28.4 小结

为了能够为之后Java EE平台上各种企业级应用服务的介绍做铺垫，我们首先在本章对JNDI进行“开刀”，首先一起简单回顾了JNDI相关内容，然后详细介绍了Spring框架为JNDI提供的集成服务。接下来，我们将进入JMS的领地，看一下Spring框架都为JMS做了哪些。

第 29 章

Spring框架对JMS的集成

本章内容

- 说说JMS的身世
- 使用JMS API进行应用开发的传统套路
- Spring改进后的JMS实战格斗术

29.1 说说 JMS 的身世

在JMS横空出世之前，存在多种MOM（Message-Oriented Middleware，面向消息中间件）产品。在企业信息系统中使用这些MOM，通常不得不使用各个MOM产品特定的访问API，可能有C语言实现的，也可能有C++或者Java语言实现的。为了能够让Java平台上使用消息机制开发的应用程序有一个统一的访问不同MOM产品的方式，JMS规范（JSR 914）就此诞生了！

JMS是一套API接口规范。因为各种MOM产品的功能特性可能多种多样，JMS规范在规范本身的实现复杂度与功能支持的广度上做了一个权衡，使得开发复杂的企业级消息应用应保无忧。JMS规范有两个主要版本，即JMS 1.02和JMS 1.1。在JMS 1.02规范中定义了如下两种明确的消息域模型（Messaging Domain）。

- 点对点模式（Point-to-Point），简称PTP模式。消息发送者将消息发送到指定的消息队列（Queue），消息接收者也是从同一消息队列中顺序获取数据，并对获取的消息进行处理。这种模式类似于多线程环境下的“生产者-消费者”场景。
- 发布订阅模式（Publish/Subscribe），即Pub/Sub模式。消息发送者将消息发送到指定的消息主题（Topic），多个消息接收者可以从相应主题中获取某个相同消息的拷贝并进行处理。这种模式很接近生活中的报纸和杂志订阅的场景：不同的杂志社/报社就相当于消息发送者，他们发行的期刊就相当于不同的主题。如果我们订阅了其中某些主题（期刊报纸），那么，每个人可以获得同一主题（报纸期刊）的一份副本（消息），然后每个人怎么读这些消息那就因人（消息接收者）而异了。

JMS 1.02规范为这两种消息域模型分别定义了两套独立的消息访问API接口设计。但JMS 1.1之后更倾向于采用较为通用的消息访问APJ接口，也就是说，可以使用一套通用的API访问现有的两种消息域模型。如果是新开发的系统，应该尽量使用JMS 1.1之后提出的新的通用访问API，因为将来可能不赞成使用之前JMS 1.02规范中为每种消息域模型单独提供的API接口设计。

一个典型的JMS应用通常由以下几个部分组成。

- JMS类型的客户端程序（JMS Client）。使用Java语言编写的用于发送或者接收消息的程序，这实际上也正是我们开发人员所要实现的东西。

- 非JMS类型的客户端程序（non-JMS Client）。使用相应消息系统或者产品提供的本地API，而不是JMS API来进行消息访问的程序。在JMS规范没有诞生之前，这种类型的客户端应该是比较常见的。
- 消息（Message）。作为消息应用，传递的当然是各种消息。JMS规定了5种消息类型，即`StreamMessage`、`BytesMessage`、`MapMessage`、`TextMessage`以及`ObjectMessage`。我们可以根据具体应用场景选择处理合适类型的消息。
- JMS Provider。JMS规范指定了一系列的API接口，而加入JMS规范的各种MOM产品，需要根据规范提供特定于它们自身产品的JMS API接口实现，以及各种相关的服务（比如消息的存储转发等）。这些特定于MOM产品的JMS API接口实现以及相关功能实现就称之为JMS Provider。现有的实现有Apache ActiveMQ（http://activemq.apache.org/）、JBoss Messaging（http://www.jboss.org/jbossmessaging/）以及IBM WebSphere MQ（http://www -306.ibm.com/software/integration/wmq/）等。
- 受管理对象（Administered Object）。为了向不同JMS类型客户端程序屏蔽各种JMS Provider之间的差异性，系统管理员可以将某些特定类型的对象，通过JMS Provider提供的管理工具提前配置到系统中。这些提前配置到系统然后再交给JMS类型客户端使用的对象就叫做受管理对象。受管理对象有如下两种类型。
  - `ConnectionFactory`。客户端需要通过`ConnectonFactory`获取相应的`Connection`，并创建其他相关对象才能访问消息系统中的消息。
  - `Destination`。即消息发送或者接收的目的地。依消息域模型的不同，`Destination`可以划分为更加具体的消息队列（Queue）和消息主题（Topic）两种类型。

通常，受管理对象被绑定到相应的JNDI服务。这样，客户端只需要通过JNDI查找指定的接口类型并使用即可，根本不需要知道这些受管理对象所对应的具体实现到底是什么类型。临时替换其他类型的JMS Provider，只需要重新配置受管理对象并绑定到JNDI，而这些不会对客户端造成任何影响。所以，JNDI所提供的“隔离层”作用在这里可见一斑。

使用JMS可以帮助我们构建低耦合且极具灵活性的应用系统，从大的方面讲，我们可以使用JMS集成各种遗留系统，或者搭建现有系统与新开发的系统之间的通信桥梁，比如各种金融或者信贷系统。之前可能多使用MainFrame架构，而新的应用可能构建PC上，那通过JMS可以很好地集成这些不同类型的系统并使它们共同工作；从小的方面讲，JMS同样可以用于系统中各个组件之间的交互和通信。不过，因为引入了分布式概念，在这种场景下使用JMS需要确保其必要性。

应该说，JMS在Java EE平台还是比较令人瞩目的，只不过，可能我们平常开发中很少用到，所以，大多数人对其都比较陌生。希望以上的简单介绍能够带你进入到JMS的世界中来。下面我们将剖析Spring对JMS的各种支持，首先了解JMS还是十分必要的。

> **提示**　要了解JMS，最好的做法就是通读一遍JMS规范（JMS Specification）文档。短短140页的内容足以让我们了解JMS的来龙去脉。当然，这并非是理解余下内容的必要条件。不过，要是需要使用JMS开发系统的话，还是建议通读一遍JMS规范定义文档。

29.2　使用 JMS API 进行应用开发的传统套路

无论是JMS规范还是有关介绍JMS的大部分文章，它们所给出的代码示例大都是使用JMS原始API

进行编写的。这当然有助于我们近距离地接触JMS API，但实际工作中要是同样按照这一套路走下来的话，或许我们就该反思一下了。因为实例的演示和实际的开发终究会有距离的。对于JMS的使用来说，情况也是如此。

对于初次使用JMS API开发应用的人来说，JMS API的使用看起来可能略显复杂。不过，稍加留意，我们就会发现其中是有规律可循的，不信？那不妨来看一下通常是如何实现消息发送功能的，见代码清单29-1。

代码清单29-1 使用JMS原始API进行消息发送的代码示例

```
ConnectionFactory connectionFactory = lookupCFViaJNDI();
Destination dest = lookupDestViaJNDI();

Connection con = null;
Session session = null;
try
{
    con = connectionFactory.createConnection();
    session = con.createSession(false, Session.AUTO_ACKNOWLEDGE);
    MessageProducer messageProducer = session.createProducer(dest);

    TextMessage message = session.createTextMessage();
    message.setText("Hi");
    messageProducer.send(message);

    messageProducer.close();
    session.close();
}
catch(JMSException e)
{
    e.printStackTrace();// 不要这样做
}
finally
{
    if(con != null)
    {
        try
        {
            con.close();
        } catch (JMSException e) {
            e.printStackTrace(); // 不要这样做
        }
    }
}
```


如果我们改用自然语言来描述这一过程，那么步骤如下所述。

(1) 首先，从JNDI获取JMS的受管理对象，即ConnectionFactory和稍后要用到的一个或者多个Destination引用。

(2) 使用获取的ConnectionFactory创建发送消息用的Connection。

(3) 使用Connection创建发送消息用的Session。

(4) 使用Session创建发送消息用的MessageProducer。

(5) 使用Session创建将被发送的消息。

(6) 调用MessageProducer发送创建完的消息到指定的Destination。

(7) 最后做资源清理，包括关闭MessageProducer、Session以及Connection。

而且对于接收消息的功能实现来说，整个步骤其实也很相似。唯一不同的就是，现在要使用Session创建接收消息的MessageConsumer，并调用Connection的start()方法开始传送消息以便接收并处理它。而像从JNDI获取相关受管理对象，创建Connection，创建Session，以及最后的资源清理这些步骤，几乎任何情况下都没有太大差别。说了这么多，我想你已经猜到Spring将对JMS API的使用做什么样的改进了。对，依然采用模板方法模式对JMS API的使用进行适度的封装，避免每次都重复编写同样的代码，以及可能存在的资源泄漏等容易出现的问题。

JMS从整体上来说是很成功的，只不过因为过于接近底层，从而导致在实际开发过程中使用过于烦琐，除此之外，其自身设计上还有一个问题，那就是对异常处理的设定不当。JMS规范虽然设计了一套以JMSException为首的异常类体系，但它们都归属于checked exception，对于消息的发送和接收过程中出现的异常，我们难道真的能够有效的处理吗？答案显然是否定的，网络出现问题，程序只能无可奈何，所以，JMSException最初更应该设计为unchecked exception，还好，现在Spring对这一问题也给予了适度的关注。

以上东西说的再多也都只是“温故”，而我们的实际目的却是“知新”，所以，还是让我们赶快看一下Spring都为JMS提供了哪些便利性支持吧！

29.3　Spring 改进后的 JMS 实战格斗术

29.3.1　消息发送和同步接收

org.springframework.jms.core.JmsTemplate是Spring框架为JMS消息发送以及同步消息接收场景提供的核心支持。该类以模板方法模式对JMS API使用上的传统套路进行了封装，很好地避免了各种相关代码的重复和散落。

1. JmsTemplate亲密接触

JmsTemplate唯一必须依赖的就是一个JMS的ConnectionFactory。无论通过什么方式满足这一条件（通过JNDI获取也好，本地实例引用也好），只要这一条件得以满足，JmsTemplate即可马上进入工作状态，如下所示：

```
ConnectionFactory cf = ...;
JmsTemplate jmsTemplate = new JmsTemplate(cf);
...
```

这与JdbcTemplate需要一个DataSource是同样的道理。

JmsTemplate的核心模板方法定义的简单抽象如代码清单29-2所示。

代码清单29-2　JmsTemplate核心模板方法原型代码示例

```
public Object execute(SessionCallback action, boolean startConnection) throws JmsException{
    try
    {
        // 获取相应的connection
        // 创建相应的session
        return action.doInJms(sessionToUse);
    }
    catch(JMSException ex)
    {
        // 转译并抛出相应异常……
    }
    finally
    {
```

```
        // 清理相应资源
    }
}
```

该方法通过SessionCallback回调接口为用户提供自定义逻辑介入点，而资源管理等一系列其他问题则由当前模板方法统一管理。startConnection参数告知当前模板方法，是否要调用Connection的start()方法开始传送消息。对于发送消息的操作来说，该操作不是必须的。如果当前要处理的是消息接收，那startConnection参数应该给予true值。鉴于startConnection可以区分当前方法用于消息发送和接收操作的性质，JmsTemplate还提供了一个简化版的execute模板方法实现，如下所示：

```
public Object execute(SessionCallback action) throws JmsException {
    return execute(action, false);
}
```

也就是说，默认情况下，如果调用该模板方法进行消息处理，是不调用Connection的start()方法的。确切地讲，该简化版模板方法只能用在发送消息的时候。

JmsTemplate的execute()方法加上SessionCallback回调接口可以给予我们最大的操作权限，但它们并不是最快捷的方式，除非我们确实要对消息发送的步骤做更多的定制。大部分情况下，直接使用JmsTemplate构建在核心模板方法上的其他模板方法要更划算一些。

JmsTemplate的模板方法大略上划分为三类，一类专门用于消息的发送，一类用于消息的同步接收，最后一类是使用QueueBrower检查消息队列的情况。下面是这三类模板方法的简单概括。

(1) **JmsTemplate的消息发送模板方法**。用于发送消息的模板方法可以进一步划分为如下三类（或者说三组）：

a) 使用ProducerCallback的模板方法。ProducerCallback接口定义如下：

```
public interface ProducerCallback {
    Object doInJms(Session session, MessageProducer producer) throws JMSException;
}
```

使用ProducerCallback回调接口的execute()模板方法是由使用SessionCallback回调接口的execute()模板方法发展而来。使用execute(SessionCallback)，我们需要在SessionCallback中，自己创建发送消息用的MessageProducer实例并在合适的时机关闭它，如代码清单29-3所示。

代码清单29-3　结合SessionCallback使用的JmsTemplate代码示例

```
jmsTemplate.execute(new SessionCallback(){

    public Object doInJms(Session session) throws JMSException {
        ObjectMessage requestMessage = session.createObjectMessage();
        MessageProducer producer = createProducer(session, destination);
        try {
            producer.send(requestMessage);
        }
        finally {
            JmsUtils.closeMessageProducer(producer);
        }
        return null;
    }});
```

29

而使用execute(ProduerCallback)方法则不需要自己来管理MessageProducer实例的创建和关闭。在ProducerCallback中，直接使用模板方法已经创建好的MessageProducer实例即可，如以下代码所示：

```
jmsTemplate.execute(new ProducerCallback(){
        public Object doInJms(Session session, MessageProducer producer) throws JMSException {

        Message message = session.createObjectMessage();
        producer.send(destination,message);
        return null;
    }});
```

b) 使用MessageCreator的模板方法。通常，JmsTemplate中以send(..)命名的模板方法可以接受MessageCreator作为参数。这里的MessageCreator将负责发送的消息的创建和相关处理，send(..)方法最终只需要直接发送MessageCreator创建完的消息即可，如以下代码所示：

```
jmsTemplate.send(new MessageCreator(){
    public Message createMessage(Session session) throws JMSException {
        ObjectMessage message = session.createObjectMessage();
        // 或者 message = session.createXXXMessage();
        ...
        return message;
    }});
```

send(..)方法除了可以接受MessageCreator作为参数，还可以接受String型或者Destination类型参数，用于标志消息发送的Destination。如果没有为send(..)方法指定消息发送的Destination相关信息，消息默认将被发送到为JmsTemplate指定的默认Destination。

c) 使用MessageConverter的模板方法。以convertAndSend(..)命名的多个方法都是使用MessageConverter的模板方法。MessageConverter在这里不是这些模板方法的方法参数，它是JmsTemplate使用的一个Helper类，负责具体消息类型与对象类型之间的相互转换。下面是这个接口的定义：

```
public interface MessageConverter {
    Message toMessage(Object object, Session session) throws JMSException,➥
    MessageConversionException;
    Object fromMessage(Message message) throws JMSException, MessageConversionException;
}
```

toMessage()方法负责将相应的对象转换为JMS的Message以便发送，而fromMessage()则负责将JMS的Message转型为应用程序所使用的对象类型，所以，convertAndSend(..)这类模板方法可以接受任何类型的对象（String类型、Map类型以及Object类型等）。MessageConverter将负责把这些对象转换为合适的Message类型发送，比如：

```
jmsTemplate.convertAndSend("text message");
// 或者
jmsTemplate.convertAndSend(new AnyBean(..));
```

这显然要比使用MessageCreator自己创建JMS的Message实例简洁得多。

JmsTemplate 默认使用的 MessageConverter 实现为 org.springframework.jms.support.converter.SimpleMessageConverter，它可以满足String类型、byte数组、Map以及可序列化对象（Serializable）与JMS对应Message之间的相互转换。如果我们需要支持更多类型，可以实现自己的MessageConverter，然后通过“messageConverter”属性设置给JmsTemplate并启用它。

convertAndSend(..)系列模板方法还有一个特性就是，我们可以同时指定一个MessagePostProcessor。通过它，我们可以在消息发送之前对消息做进一步的后处理，如下代码给出了这种情况下的代码示例：

```
Object yourMessage = ...;

jmsTemplate.convertAndSend(yourMessage,new MessagePostProcessor(){
    public Message postProcessMessage(Message message) throws JMSException {
        message.setStringProperty("STOCK-MARKET", "NASDAQ");
        return message;
    }});
```

我们在消息发送之前为消息设置了相应的Property，以便接收方可以根据该Property使用相应的Message Selector选取/过滤要接收的消息。

每组中的各种重载方法的详细情况，可以参考JmsTemplate的Javadoc，这里就不做一一罗列了。

(2) **JmsTemplate的同步消息接收模板方法。**用于同步消息接收的模板方法。其内部同样有不同的派别，好在不管派别如何，都是为了简化我们的开发过程。下面是各个派别的情况。

a) 直接同步接收的模板方法。以receive(..)命名的模板方法是最直接也是最简单的同步消息接收方法。如果要指定参数的话，receive(..)方法只接受String类型或者Destination类型的参数作为消息接收的Destination。如果不指定任何参数，那么receive()方法将直接从JmsTemplate的默认Destination中接收消息。

b) 可以指定MessageSelector表达式的模板方法。JMS提供的Message Selector[①]机能，可以让消息接收方指定接收符合某种条件的消息。JmsTemplate中以receiveSelected(..)命名的模板方法，允许我们传入selector表达式来启用Message Selector机能，比如：

```
Message receivedMessage = jmsTemplate.receiveSelected("STOCK-MARKET='NASDAQ'");
// 处理接收到的信息……
```

有关Message Selector可以使用的表达式语法，可以参考JMS规范文档（JMS Specification），这里不再赘述。

c) 使用MessageConverter的模板方法。与消息发送模板方法相对的，用于同步消息接收的也有一组使用MessageConverter的模板方法。这些模板方法以receiveAndConvert(..)命名。如果说receive()和receiveSelected(..)方法所返回的都是原始的JMS Message类型的话，现在的receiveAndConvert(..)返回的则是由Message转型后的对象类型，如下所示：

```
String textMessage = (String)jmsTemplate.receiveAndConvert();
Map mapMessage = (Map)jmsTemplate().receiveAndConvert();
Object objectMessage = jmsTemplate().receiveAndConvert();
...
```

至于MessageConverter，我想就不用多说了吧？

d) 结合selector表达式和MessageConverter的模板方法。如果我们既想使用MessageConverter，又想同时使用Message Selector机能，那么，以receiveSelectedAndConvert(..)命名的这组模板方法就是我们想要的了。在使用MessageConverter的模板方法实现逻辑上添加selector表达式的指定，就是receiveSelectedAndConvert(..)的奥秘，一切都是为了提供更多更便捷的服务。

JmsTemplate提供的所有同步消息接收模板方法都没有提供相应参数，用于指定消息接收的超时时间（Timeout）。默认情况下，它们都将无限期地等待，这将阻塞当前调用线程。在实际使用中，最好是根据场景，为消息的接收指定一个合适的超时时间，可以通过JmsTemplate的receiveTimeout属性设定。

① 有关Message Selector的详细信息，可以参考JMS 1.1规范的3.8节。

(3) **检查消息队列情况的模板方法。** JMS规范提供的QueueBrowser允许我们查看某一消息队列的情况。不过，"只能看不能碰"。JmsTemplate同样为这种情况下的API使用进行了封装，封装后的模板方法可以简单划分为如下两组。

a) 不使用MessageSelector的模板方法。JmsTemplate中以browse命名的模板方法可以接受Brow- serCallback作为回调接口，该接口定义如下：

```
public interface BrowserCallback {
    Object doInJms(Session session, QueueBrowser browser) throws JMSException;
}
```

我们可以提供自定义的BrowserCallback实现，在其中使用模板方法公开给我们的QueueBrowser进行消息队列的检查。至于QueueBrowser如何创建和关闭，那就由browse(..)模板方法操心就行了。

b) 使用MessageSelector的模板方法。重载后的browseSelected(..)这组模板方法较之browse(..)那组模板方法多了一个功能，那就是可以同时指定一个selector表达式。至于差别，我想就不需要多提了吧？

无论是用于消息发送还是用于同步消息接收的模板方法，它们当中都会有一个不需要指定Destination的重载方法。这得益于JmsTemplate的defaultDestination属性所指定的默认Destination。如果系统中有多个消息被发送到同一个Destination，或者从同一个Destination接收，那么将该Destination设置为所使用的JmsTemplate的defaultDestination将是合适的做法。甚至，即使我们在发送消息或者接收消息的时候指定了要使用的Destination，但如果指定的Destination找不到的话，JmsTemplate也会使用defaultDestination指定的默认Destination作为"后备力量"。所以，通常情况下，通过defaultDestination为JmsTemplat设置一个默认的Destination还是有必要的。

为JmsTemplate指定默认的Destination并不意味着我们从此以后就只能向这个默认的Destination发送消息或者从它那里同步接收消息，我们依然可以在调用消息发送或者接收方法的时候指定其他的Destination。这时，指定Destination的方式可以有两种，一种是指定直接的Destination类型实例，另一种是以String形式给出。对于后者来说，JmsTemplate需要DestinationResolver的帮忙，把String类型的Destination名称转换为具体的Destination实例。DestinationResolver的定义如下所示：

```
public interface DestinationResolver {
    Destination resolveDestinationName(Session session, String destinationName, boolean➥
    pubSubDomain) throws JMSException;
}
```

DestinationResolver的实现类只需要根据destinationName去获取对应的Destination实例即可。在Spring框架内部，DestinationResolver主要有如下三个可用的实现类。

- **org.springframework.jms.support.destination.DynamicDestinationResolver。** 这是JmsTemplate默认使用的DestinationResolver实现，它将根据destinationName创建动态的Destination实例。
- **org.springframework.jms.support.destination.JndiDestinationResolver。** 对于事先配置到JNDI的Destinations，我们需要使用JndiDestinationResolver。这个时候，就可以通过设置JmsTemplate的destinationResolver属性，以某一JndiDestinationResolver替换默认的DynamicDestinationResolver。

- **org.springframework.jms.support.destination.BeanFactoryDestinationResolver**。如果可以将Destination以bean定义的形式添加的Spring的IoC容器的话，BeanFactory-DestinationResolver将可以保证根据destinationName去容器中获取相同beanName的Destination实例。

DynamicDestinationResolver和JndiDestinationResolver是比较常用的Destination-Resolver实现类，这可以因JMS Provider的不同以及应用的场景需求而发生变化。不过，我想，现有的DestinationResolver实现类已经足够用了。如果还有什么特殊需求的话，实现Destina-tionResolver也不难，只有一个接口方法要实现而已。

JmsTemplate最引入注目的，当然就是它提供的各式各样服务周到的模板方法。不过，这些模板方法指定的参数都比较有限，无法做更细粒度的功能定制。这不是说JmsTemplate设计上有问题，实际上，那些在消息处理期间比较通用的功能或者说行为，都是通过JmsTemplate定义的相应属性来进行控制的，如receiveTimeout、defaultDestination等属性，我们已经认识过了。表29-1中是其他几个可以对JmsTemplate行为进行定制的属性。

表29-1 JmsTemplate部分属性说明

属　性	默认值	说　明
pubSubNoLocal	false	是否禁止传输当前Connection发布的消息
pubSubDomain	false	默认值为false，即使用点对点消息域模型，JmsTemplate可以根据pubSubDomain值来决定Destination的具体类型。当然，在JMS 1.1中，这通常是不必要的
explicitQosEnabled	false	标志是否需要启用指定的deliveryMode、priority和timeToLive对消息传输行为进行定制。默认值是忽略这些设定。如果将explicitQosEnabled的值设置为true，那么，deliveryMode，priority和timeToLive对应的属性值将被应用到消息发送过程中

JmsTemplate看起来可能比较庞大，但实际上，其实现原理并不复杂，尤其是，Spring框架内使用模板方法模式加上相应回调接口的API封装方式，我们已经再熟悉不过了。所以，如果你要了解有关JmsTemplate的更多秘密，不妨自己再发掘一下。

注意 JmsTemplate主要面向的是JMS 1.1规范。如果你不得不依然使用JMS 1.02规范进行开发，那么，使用org.springframework.jms.core.JmsTemplate102即可。Spring为JMS提供的各种装备，对应JMS 1.02规范的，都较之JMS 1.1的相关类名后缀102，比如，除了JmsTemplate102，还有SimpleMessageConverter102等。

2. 同步消息处理场景浅析

外汇交易系统中大量使用JMS进行交互，比如系统与银行之间的交互，不过，这涉及银行内部的一些处理细节，通常我们是无法了解的，所以，我们改变以往的策略，转而构建一个简化版的JMS同步消息处理场景。在这个场景中，服务器端提供通用的用户验证服务，在客户端以JMS消息的形式发送验证请求之后，服务器端将接收这些验证请求并进行处理，一旦处理完毕，则将验证结果同样以JMS消息的形式发送给请求的客户端，客户端只有接收到通过验证后的反馈结果才能进行其他操作。整个情况如图29-1所示。

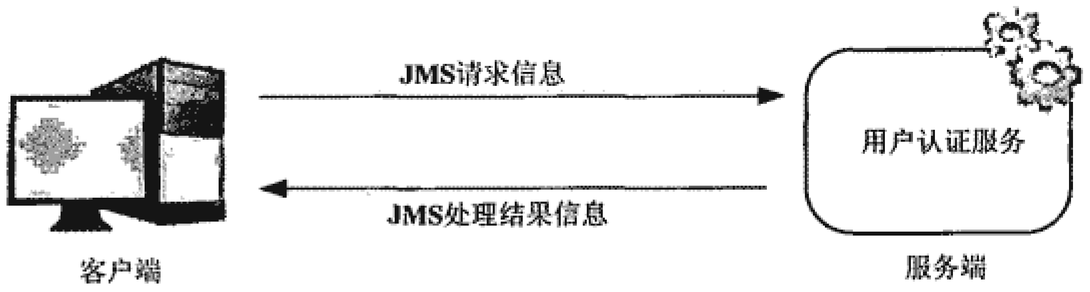

图29-1　验证服务JMS同步消息交互原型图

虽然整个原型图（图29-1）看起来比较简单，但如果你有心的话，完全可以在这幅图的基础上扩展出更加实用的系统架构图。这丝毫没有夸张之意，对于我们所开发的Java EE应用程序来说如是，对于网络游戏服务器端的架构也同样如此。

从整个原型图（图29-1）上来看，客户端发送验证请求的JMS消息之后，需要同步等待服务端的验证结果，也就是说，客户端发送验证请求的消息之后，会直接同步接收验证的结果消息，只有接收到验证通过的消息之后，客户端才允许进行后继的处理工作。这其实就是我们将要一起来实现的那个同步消息处理场景，而至于服务端如何处理请求消息并返回验证结果，那是稍后在介绍Spring的异步消息支持的时候才需要操心的事情。现在，我们要考虑的是如何实现客户端的同步消息发送和接收的问题。

我们的初步设想是这样的，为验证请求信息预先配置一个专用消息队列，暂且称之为authRequestQueue，同时，为验证结果信息也同样配置一个专用消息队列，不妨称之为authResponseQueue。客户端将验证请求信息发送到authRequestQueue之后，就可以马上进入同步消息接收状态，从authResponseQueue中接收返回消息。将我们的原型图进一步细化之后，就可以得到如图29-2所示的示意图。

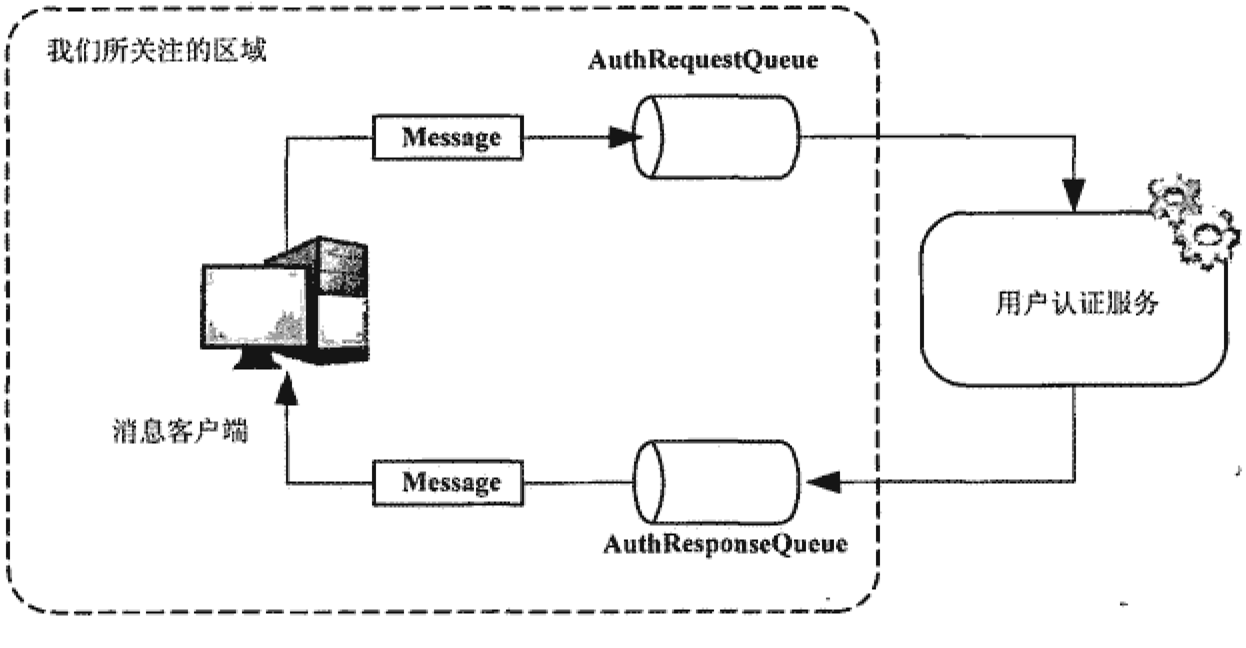

图29-2　初步细化后的实现原型图

应该说，我们的设想以及所看到的结果是很漂亮的。不过，它并不完美，如果说，这一场景中只有一个或者几个客户端需要验证服务，这样的设想实现是比较合适的，它在性能与队列管理复杂度之间取得了一个很好的平衡，但是，随着连接客户端的增加，我们是否要为每个客户端都维护一对儿

authRequestQueue和authResponseQueue呢？虽然这么做可以保证性能不受损失，但消息队列的管理复杂度将直线上升，这是不可取的。看来，我们得找其他方案。

既然不想因消息队列的数量增加来引入过多的管理复杂度，我们就不引入多余的消息队列，可以使用单一的消息队列authRequestQueue来接收所有客户端发送的验证请求信息，并且使用同样是单一的一个消息队列authResponseQueue来返回验证后的结果信息。不过，为了保证客户端只接收到自己对应的验证信息，我们需要一个Message Selector来帮助客户端过滤掉它不感兴趣的那部分。现在，我们有了另一副实现的设想图（见图29-3）。

看起来，这比之前的设想要更加丰满和成熟一些，那么，它是否就完美了呢？事实上，也不尽然，使用Message Selector会引入一定的性能损耗。不过，当前的方案还是可以使用的，姑且先按照这个方案实现客户端如代码清单29-4所示。

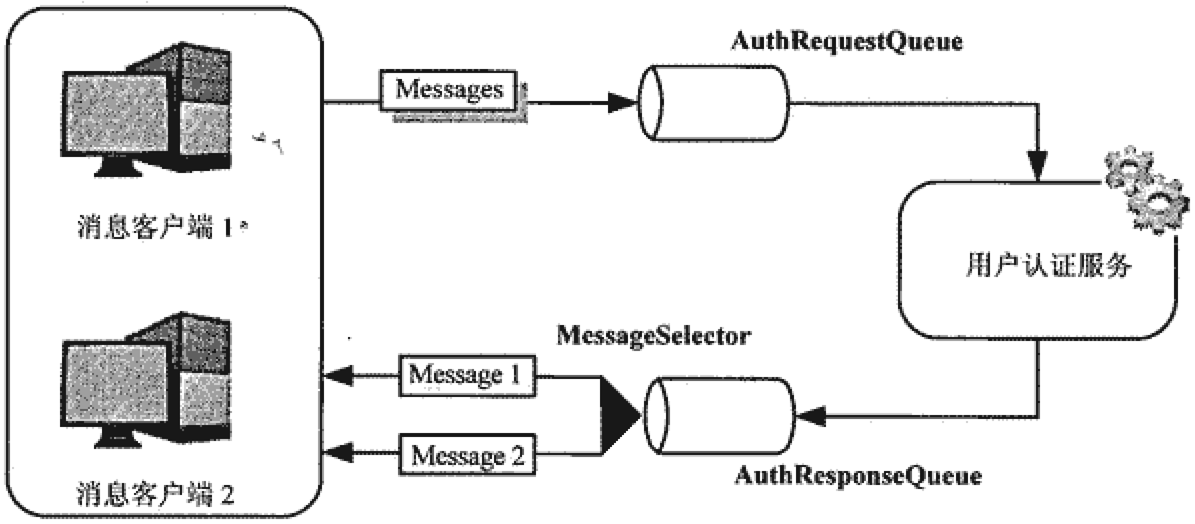

图29-3 进一步细化后的实现原型图

代码清单29-4 使用特定单一队列实现消息发送和同步接收功能的代码示例

```
final AuthRequest authReq = new AuthRequest();
authReq.setUserId("...");
authReq.setPassword("...");
...

String messageSelector = SystemMessageProperties.REQUEST_ID + " ='" +requestId+ "'";

getJmsTemplate().convertAndSend(getRequestDest(), authReq, new MessagePostProcessor(){
    public Message postProcessMessage(Message message) throws JMSException {
        message.setStringProperty(SystemMessageProperties.REQUEST_ID, requestId);
        return message;
    }});

Message responseMessage = getJmsTemplate().receiveSelected(getResponseDest(), messageSelector);
processMessage(responseMessage);
```

SystemMessageProperties.REQUEST_ID只是一个常量，用于标志Message Selector使用的JMS Message的Property。要以当前这个方案实现客户端的主要理由是，它基本上反映了JmsTemplate的绝大多数的技能，不是吗？

既然使用单一队列加MessageSelector的方案可能因为性能问题不能够在某些比较严格的场景下使

用，我们现在给出最后一种方案，即使用Temporary Destination。我们让每个客户端在发送验证请求消息之前，首先创建一个临时的消息队列（Temporary Queue），然后将这一创建好的临时消息队列设置为当前要发送消息的JMSReplyTo。在服务端处理完请求消息之后，就可以根据当前消息的JMSReplyTo，将验证信息返回到客户端指定的临时消息队列中，客户端直接从自己创建的临时消息队列中接收消息即可。现在，整个同步消息发送和接收的设想图就如图29-4所示。

Temporary Destination的存活时间通常与创建它的Connection相挂钩。为了保证这一点儿，我们通常可以使用execute(SessionCallback)模板方法。代码清单29-5是我们的实现代码（或者其他方式，只要保证消息的发送和接收与使用的Temporary Destination处于同一Connection下即可）。代码清单29-5是我们的实现代码。

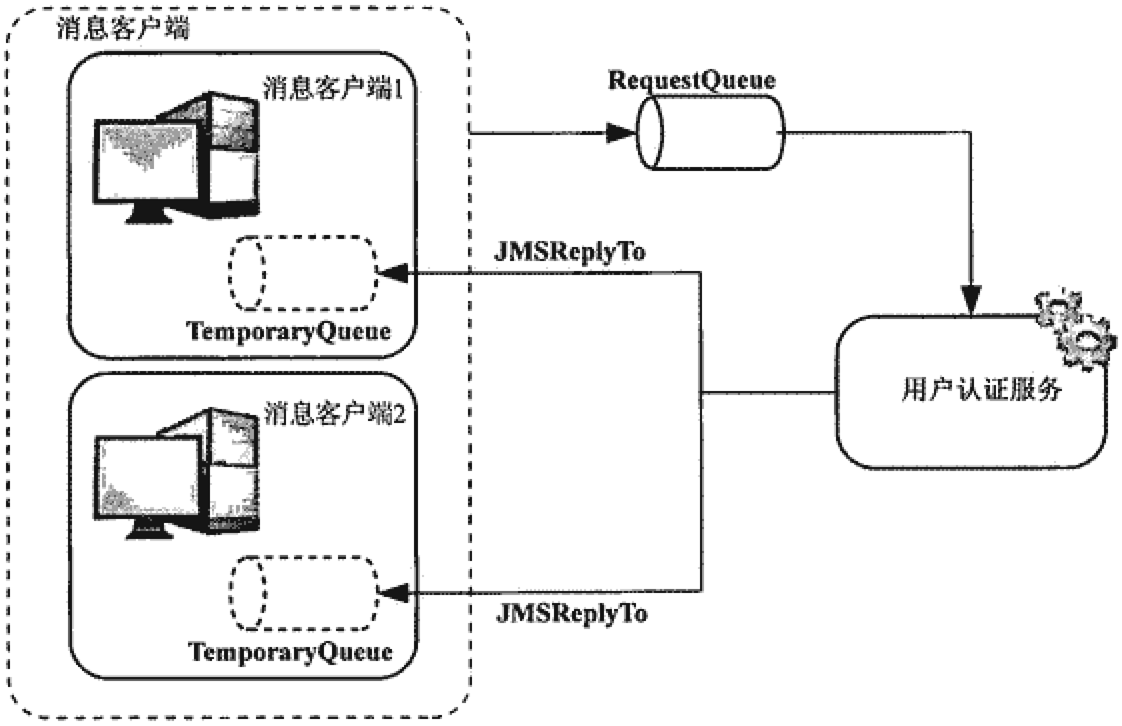

图29-4　使用临时消息队列处理同步消息接收的场景示意图

代码清单29-5　使用Temporary Queue实现同步消息接收的代码示例

```
final AuthRequest authReq = new AuthRequest();
authReq.setUserId("..");
authReq.setPassword("..");

getJmsTemplate().execute(new SessionCallback(){
    public Object doInJms(Session session) throws JMSException {
        ObjectMessage message = session.createObjectMessage();
        message.setObject(authReq);
        TemporaryQueue responseDestination = session.createTemporaryQueue();
        message.setJMSReplyTo(responseDestination);

        MessageProducer producer = session.createProducer(getRequestDest());
        try
        {
            producer.send(message);
        }
        finally
        {
```

```
            JmsUtils.closeMessageProducer(producer);
        }

        MessageConsumer consumer = session.createConsumer(responseDestination);
        try
        {
            Message receivedMessage = consumer.receive(10000L);
            processMessage(receivedMessage);
        }
        finally
        {
            JmsUtils.closeMessageConsumer(consumer);
        }
        return null;
}}, true);
```

使用Temporary Queue，可以避免事先创建大量的Destination，同时又不用因从单一消息队列接收消息而必须使用Message Selector。可谓一举两得，何乐而不为呢？

不过，世事无绝对，到底要采用哪种方式处理消息发送和同步的消息接收，需要你根据自己应用程序的当前场景来决定。

> **注意** 以上所描述的每一种使用场景都不是普遍适用的，我们需要根据具体的应用场景来权衡利弊，并最终决定哪一种才是最适合当前应用场景的处理方式。

29.3.2 异步消息接收

在一个典型的J2EE场景中，处理异步消息的标准组件通常就是一个消息驱动Bean（Message-Driven Bean，简称MDB）。不过，要使用消息驱动Bean来处理异步消息通信的话，我们的应用就不得不依赖于EJB容器（EJB Container）。为了避免进行异步消息处理的应用程序对EJB容器的依赖，Spring提供了`MessageListenerContainer`。借助于`MessageListenerContainer`的支持，我们现在可以创建消息驱动POJO（Message-Driven POJO，简称MDP）来处理异步消息。不过，在开始创建真正可以运行的消息驱动POJO之前，我们有必要先了解一下`MessageListenerContainer`背后到底隐藏着什么样的奥秘。

> **小心** Spring提供的异步消息支持是在2.0之后引入的。如果你不得不使用Spring 1.x版本，那么只能获得`JmsTemplate`同步消息发送和接收的好处。

29

1. 了解MessageListenerContainer

`MessageListenerContainer`之所以叫做`MessageListenerContainer`，是因为用于处理消息的消息驱动POJO必须实现`MessageListener`接口，而`MessageListenerContainer`本身就负责管理这些实现了`MessageListener`的消息驱动POJO对象。这其实就跟消息驱动Bean的生存环境一样，所有的消息驱动Bean都需要实现`MessageListener`接口，而至于从哪里接收消息，接收到的消息最终如何分配给具体的消息驱动Bean来处理，那就完全是EJB容器的责任了。不负责任地讲，`MessageListenerContainer`实际上是接管了EJB容器在异步消息接收处理领域的部分职责。好在这种接管行为，确实为应用程序带来了更轻量级的异步消息处理方案，这也就是为什么消息驱动POJO现在是如此的受人欢迎。

往简单了讲，MessageListenerContainer就有两个职责：

- 负责到指定的Destination接收符合处理条件的消息；
- 将接收到的消息通过某种策略转发给指定类型的MessageListener实现类来处理。

不过，实际的MessageListenerContainer所要关注的不只是这两点。既然要将应用程序彻底地从JMS API的使用和管理中解放出来，真正的MessageListenerContainer实现类还需要关注Connection、Session等一系列对象的生命周期管理，异常的处理，甚至消息处理过程中涉及的本地事务与全局事务相关的种种关注点。说白了，MessageListenerContainer把“脏活儿累活儿”全部揽到自己的身上，我们只要在自己的MessageListener实现类中处理已经分给我们的Message就成了。

Spring框架默认提供了三种MessageListenerContainer实现类，如下所述。

(1) **org.springframework.jms.listener.SimpleMessageListenerContainer**。

SimpleMessageListenerContainer是三种MessageListenerContainer实现类中最简单的，它采用JMS规范提供的标准异步消息接收方式来接收消息，即通过MessageConsumer.SetMessageListener(..)设定用于异步消息处理的MessageListener。为了能够扩展处理性能，SimpleMessageListenerContainer可以在初始化的时候同时指定创建多个Session用于消息处理。如果将SimpleMessageListenerContainer的处理逻辑简化一下，那么代码清单29-6所示的原型代码可能会让你看起来更加亲切一些。

代码清单29-6　SimpleMessageListenerContainer原型代码示例

```
int coucurrentCount = 5;
Destination destination = // 接收消息使用到的Destination
MessageListener messageListener = // 相应的MessageListener实现

// 最好是注入相应的ConnectionFactory实例
ConnectionFactory connectionFactory = new ActiveMQConnectionFactory();
Connection connection = connectionFactory.createConnection();
Session[] concurrentSessions = new Session[coucurrentCount];
MessageConsumer[] concurrentConsumers = new MessageConsumer[coucurrentCount];

for(int i=0;i<coucurrentCount;i++)
{
    concurrentSessions[i] = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
    concurrentConsumers[i] = concurrentSessions[i].createConsumer(destination);
    concurrentConsumers[i].setMessageListener(messageListener);
}
connection.start();
```

我们的原型代码省略了异常的处理，同时许多可配置项也都写死到代码了。不过，使用SimpleMessageListenerContainer，这些都不再是问题。

SimpleMessageListenerContainer允许我们指定一个TaskExecutor用于消息处理的调度。如果没有明确指定任何TaskExecutor的话，为SimpleMessageListenerContainer指定的MessageListener将在JMS Provider的当前线程中处理消息，直到处理完成之前，将一直阻塞JMS Provider的当前线程。所以，SimpleMessageListenerContainer在性能扩展方面的发挥余地不是很大。

> **注意**　有关TaskExecutor的详细情况和可用的实现类型，可以参考稍后的内容。

(2) **org.springframework.jms.listener.DefaultMessageListenerContainer**。

虽然DefaultMessageListenerContainer是用于异步消息处理的MessageListenerContainer

实现类，但它本身的实现原理却有些“背道而驰”。实际上，DefaultMessageListenerContainer在接收指定Destination的消息的时候，不像SimpleMessageListenerContainer那样使用JMS规范的标准装备，而是反其道而行之，转而使用同步的接收方法。这当然有些“大逆不道”的味道，但这样的处理方式却可以给予DefaultMessageListenerContainer更大的发展空间。借助于相应的Task-Executor，DefaultMessageListenerContainer可以同时启动多个线程循环调用JMS的同步消息接收方法，而接收到的消息的处理，则完全是在自己的线程内进行，可以说互不干扰，这比依赖于JMS Provider的调度策略要灵活得多。

如果我们将DefaultMessageListenerContainer的实现原理进行简化，那么可以以代码清单29-7中的原型代码为基础，做进一步的理解。

代码清单29-7 DefaultMessageListenerContainer实现原型代码示例

```
int coucurrentCount = 5;

ConnectionFactory connectionFactory = new ActiveMQConnectionFactory();
final JmsTemplate jmsTemplate = new JmsTemplate(connectionFactory);

Runnable processor = new Runnable(){
    public void run() {
        while(containerAlive)
        {
            Message message = jmsTemplate.receive(getDestination());
            getMessageListener().onMessage(message);
        }
    }
};

Thread[] threads = new Thread[coucurrentCount];
for(Thread thread : threads)
{
    thread = new Thread(processor);
    thread.start();
}
```

当然，DefaultMessageListenerContainer不可能如此简单，不过，原理上是相似的，只不过比我们的原型做了更多抽象的工作而已。DefaultMessageListenerContainer内部的消息接收和分发不是像我们那样，每次都创建新的线程，而是将调度的工作转给了相应的TaskExeuctor。而且，DefaultMessageListenerContainer内部也不是真的使用了JmsTemplate，我们只是为了简化原型代码才引入JmsTemplate，DefaultMessageListenerContainer所做的工作要比原型所做的更多，更加严谨。

如果没有什么特殊需求的话，DefaultMessageListenerContainer将是我们使用最多的MessageListenerContainer实现。

(3) **org.springframework.jms.listener.serversession.ServerSessionMessage Listener-Container**。JMS规范中规定的ServerSessionPool SPI并非强制所有的JMS Provider都提供支持，它是可选的。所以，使用ServerSessionMessageListenerContainer之前有一点需要确认，那就是你所使用的JMS Provider是否支持ServerSessionPool。如果答案是“否”的话，DefaultMessageListener-Container才是合适的选择。

ServerSessionMessageListenerContainer是Spring使用JMS规范中“Chapter 8 JMS Application Server Facilities”一章中规定的标准API构建的MessageListenerContainer实现。如果我们想要使用

JMS Provider构建于ServerSessionPool上的各种消息处理机制的话，那么可以直接使用Server-SessionMessageListenerContainer。

所有的MessageListenerContainer都实现了org.springframework.context.Lifecycle接口，所以，只要我们将具体的MessageListenerContainer实现类添加到Spring的IoC容器，IoC容器将保证相应的MessageListenerContainer得以运行，如下所示：

```
<bean id="messageListenerContainer" class="org.springframework.jms.listener. ➥
DefaultMessageListenerContainer">
    <property name="connectionFactory" ref="connectionFactory"/>
    <property name="destination" ref="authRequestQueue"/>
    <property name="messageListener" ref="authReqListener"/>
</bean>
```

我们只为DefaultMessageListenerContainer指定了必要的配置项。如果需要做更多定制，比如替换默认的TaskExecutor等，可以参考MessageListenerContainer相应实现类的Javadoc文档。

如果我们感觉以上的内容不够简洁，让人无法容忍，那好消息是，Spring 2.5之后，为基于XSD的配置引入了jms专有的命名空间，同样的配置内容，现在可以配置如下（使用之前不要忘了把jms的命名空间声明添加到配置文件中）：

```
<jms:listener-container>
    <jms:listener destination="authRequestQueue" ref="authReqListener"/>
</jms:listener-container>
```

因为<jms:listener-container>有比较合理的默认值，所以，许多东西我们可以不做明确指定，比如connectionFactory。<jms:listener-container>实际上定义有许多可以对其行为进行定制的属性，最明了的文档当然就是spring-jms.xsd。你可以在Spring 2.5发行包的dist\resources目录下找到它。不过通常不用这么做，如果有一个支持XSD的XML编辑器，使用“Code Assist”功能就直接可以获得相应提示了。

有关MessageListenerContainer就先说到这里了。下面我们得来看一下MessageListener-Container所服务的对象了，那就是MessageListener。哦，不对，更广义上讲，是消息驱动POJO（Message-Driven POJO）。

2. 消息驱动POJO

一个POJO能够转变为一个消息驱动POJO，最主要的差别就在于它头上的那顶帽子。当然，我是指MessageListener接口。MessageListener是消息驱动POJO身份的象征，所以，消息驱动POJO必须实现MessageListener接口。好在，这是一个消息驱动POJO所需要面对的唯一限制，除此之外，无论你继承什么父类，实现什么其他的接口，那都是你的自由了。一个普通的消息驱动POJO实现看起来就像如下这么简单：

```
public class YourMDPImpl implements MessageListener {
    public void onMessage(Message msg) {
        try {
            // 处理msg
        }
        catch (JMSException e)
        {
            throw JmsUtils.convertJmsAccessException(e);
        }
    }
```

```
}
```

这看起来跟普通的对象定义没有太大差别，不是吗？如果需要其他依赖的话，为其声明依赖，然后通过IoC容器注入即可。现在只要将这个YourMDPImpl添加到容器，并与相应的MessageListenerContainer相关联，它就可以借助于MessageListenerContainer的支持，进行异步的消息处理了，如下配置所示：

```
<jms:listener-container>
    <jms:listener destination=".." ref="yourMDP"/>
</jms:listener-container>

<bean id="yourMDP" class="">
    <!--可能的依赖注入对象-->
</bean>
```

整个基于MessageListenerContainer的消息驱动POJO开发，实际上就这么简单。

在29.3.1节的第2小节中，为了演示消息的同步发送和接收，我们只实现了客户端部分的功能。现在，我们要给出服务端的实现了。在该场景中，服务端负责从authRequestQueue消息队列中异步接收客户端提交的验证请求消息，然后根据消息的内容调用服务端的相应服务对象进行验证工作。不管最终验证是否通过，服务端需要将验证结果以JMS消息的形式发送给请求的客户端。为了完成这样的功能，我们实现了如代码清单29-8所示的消息驱动POJO。

代码清单29-8　用于处理验证消息的消息驱动POJO实现代码示例

```
public class AuthRequestListener implements MessageListener {
    private JmsTemplate jmsTemplate;
    private MessageConverter messageConverter = new SimpleMessageConverter();
    private IUserAuthService userAuthService;

    public void onMessage(Message msg)
    {
        try
        {
            AuthRequest authRequest = (AuthRequest)getMessageConverter().fromMessage(msg);
            AuthResponse authResponse = getUserAuthService().processAuthRequest(authRequest);
            getJmsTemplate().convertAndSend(msg.getJMSReplyTo(),authResponse);
        }
        catch (JMSException e)
        {
            throw JmsUtils.convertJmsAccessException(e);
        }

    }

    // getter和setter方法定义……
}
```

AuthRequestListener将通过一个IUserAuthService对客户端通过JMS消息发送的验证请求进行处理，然后将验证结果再次打包成JMS消息，发送回请求的客户端。因为我们在客户端是通过创建TemporaryQueue来指定消息接收的Destination，所以，AuthRequestListener将取得的JMSReplyTo对应的值作为最终消息发送的Destination，整个过程就像代码所展示的那么简单。最后，要让AuthRequestListener能够工作，需要将其添加到容器并与MessageListenerContainer相关

联，所以可以做代码清单29-9所示的配置。

代码清单29-9　AuthRequestListener相关配置示例

```
<jms:listener-container>
    <jms:listener destination="authRequestQueue" ref="authReqListener" ➥
   selector= JMSReplyTo is NOT NULL"/>
</jms:listener-container>

<bean id="authReqListener" class="..AuthRequestListener">
    <property name="userAuthService" ref="userAuthService"/>
    <property name="jmsTemplate" ref="producerTemplate"/>
</bean>

<bean id="userAuthService" class="..UserAuthServiceImpl">
</bean>
```

有时，只戴一顶帽子或许会感觉单调，或者说，这顶帽子提供的选择比较少，那么，可以尝试一下Spring为消息驱动POJO提供的另一顶帽子，即SessionAwareMessageListener。该接口定义如下：

```
public interface SessionAwareMessageListener {
    void onMessage(Message message, Session session) throws javax.jms.JMSException;
}
```

SessionAwareMessageListener使得我们可以接触要处理的消息的同时，获得对应的Session的引用。如果我们要使用接收消息的Session做更多的工作，SessionAwareMessageListener可是不二之选哦！

Spring 提供的所有MessageListenerContainer都支持MessageListener和SessionAware-MessageListener两种类型的消息驱动POJO。所以，使用SessionAwareMessageListener，除了会使我们绑定到Spring框架的API上之外，不会有任何其他的后顾之忧。不过，是否使用SessionAwareMessageListener，最终的决定权还是在你的手里。

好啦，或许你会说，这两顶帽子我看着都不爽，能不能不要？好的，没问题！如果我们不想自己的POJO实现MessageListener和SessionAwareMessageListener两者中任一接口，那么把要实现MessageListener或者SessionAwareMessageListener接口这样的强制责任推给其他“人”就好了，而且Spring提供的MessageListenerAdapter实际上就是做这个工作的！

MessageListenerAdapter自身实现了MessageListener和SessionAwareMessageListener两个接口。所以，将它与某个MessageListenerContainer挂钩之后，MessageListenerAdapter就可以行使异步消息处理的职责。MessageListenerAdapter可以接受一个委派对象。当MessageListenerContainer调度MessageListenerAdapter进行消息处理的时候，MessageListenerAdapter实际上是把真正的处理委派给了它所持有的那个委派对象。要想我们的POJO不实现任何消息驱动POJO的相关接口，只要将它设置为MessageListenerAdapter所使用的委派对象即可。

实际上，我们完全可以将刚才的AuthRequestListener看作是原型版的MessageListener-Adapter，真正的MessageListenerAdapter所做的事情与其差不了多少。在AuthRequestListener中，IUserAuthService实际上就是我们不想实现任何消息驱动POJO的相关接口的POJO对象。为了在处理消息过程中使用到IUserAuthService的相关逻辑，我们通过MessageConverter来实现具体的Message与方法参数类型之间的转换。待IUserAuthService的处理方法完成后，再将处理结果打包成返回的消息并发送，而发送的Destination恰好是请求消息的JMSReplyTo所指定的。

MessageListenerAdapter的实现稍微复杂一些，但绝对跟AuthRequestListener十分接近，如

下所述。

- AuthRequestListener只能委派IUserAuthService类型的对象进行消息处理，而MessageListenerAdapter允许使用任意类型的委派对象，然后通过反射调用处理方法。
- AuthRequestListener使用MessageConverter进行Message类型与具体对象类型之间的转换。MessageListenerAdapter也是如此。
- AuthRequestListener使用JmsTemplate发送返回消息。MessageListenerAdapter因为有onMessage(message,session)方法公开的Session，所以，直接通过Session创建发送消息的相关对象。
- AuthRequestListener将请求消息的JMSReplyTo作为返回消息的Destination。MessageListenerAdapter只有在委派对象处理方法有返回值的情况下才这么做，如果委派对象的处理方法不返回任何值（void），那么不会发送返回信息。

通过这样的对比，你是否已经可以实现一个自己的MessageListenerAdapter了呢？不过，要真的实现一个自己的MessageListenerAdapter还是免了吧！Spring提供的MessageListenerAdapter已经够用了。

如果将我们的AuthRequestListener所实现的功能，转而使用MessageListenerAdapter来实现的话，最终的结果如代码清单29-10所示。

代码清单29-10　MessageListenerAdapter使用代码示例

```
<jms:listener-container>
    <jms:listener destination="authRequestQueue" ref="msgListener" selector="JMSReplyTo is➥
    NOT NULL"/>
</jms:listener-container>

<bean id="msgListener" class="org.springframework.jms.listener.adapter. ➥
MessageListenerAdapter">
    <property name="delegate" ref="userAuthService"></property>
    <property name="defaultListenerMethod" value="processAuthRequest"/>
</bean>

<bean id="userAuthService" class="..UserAuthServiceImpl">
</bean>
```

这里唯一需要注意的就是，我们为MessageListenerAdapter指定了委派对象为userAuth-Service的同时，还指定了其defaultListenerMethod属性，该属性是告诉MessageListener-Adapter通过反射应该调用委派对象的哪个方法。如果不明确指定defaultListenerMethod值的话，MessageListenerAdapter将使用默认值“handleMessage”作为将被调用的委派对象的处理方法名称。

> 注意　有关MessageListenerAdapter的更多定制属性，请参考该类的Javadoc文档，比如MessageListenerAdapter所使用的MessageConverter和DestinationResolver等。

29.3.3　JMS 相关异常处理

JMS规范（在7.x节）定义了一套完整的异常类层次体系，以javax.jms.JMSException为首，下

设多种标准的异常类型，比如，IllegalStateException、InvalidDestinationException、InvalidSelectorException等。从异常层次体系的设定上来说，这没太多可挑剔的，但是，整个的JMSException异常层次体系是以checked exception为基准的。这就意味着，所有JMS相关的方法要么在方法内部捕获它们并处理，要么就在每个方法声明中声明这些可能抛出的异常类型。可是，对于应用程序来说，即使是捕获了这些异常，通常情况下也无法有效地处理。将JMSException的异常体系建立在unchecked exception类型基础上要更合理一些。

Spring在原有JMSException异常层次体系的基础上提出了一套使用unchecked exception声明的异常层次体系，以JmsException为代表（注意JmsException和JMSException的区别）。下面的其他多种异常类型则直接映射到JMSException的异常体系中同名的异常类，二者名称相同，只有unchecked exception和checked exception之分。表29-2给出了它们之间详细的对比关系。

表29-2　Spring的JMS异常层次体系

Spring提供的异常类型	对应JMS规范中定义的异常类型
org.springframework.jms.JmsException	javax.jms.JMSException
org.springframework.jms.IllegalStateException	javax.jms.IllegalStateException
org.springframework.jms.InvalidClientIDException	javax.jms.InvalidClientIDException
org.springframework.jms.InvalidDestinationException	javax.jms.InvalidDestinationException
org.springframework.jms……	javax.jms……

注意　要了解更多JmsException相关异常类型，请参考Spring的Javadoc文档，这些异常类型全部定义在org.springframework.jms包下。

org.springframework.jms.support.JmsUtils工具类的convertJmsAccessException(JMSException)方法，提供了从JMS的标准异常到Spring的JMS异常的转译（translate）支持。如果我们是使用JmsTemplate或者MessageListenerContainer，那么这样的异常转译已经由它们内部搞定了。只有在不使用它们的情况下，我们才会考虑JmsUtils的这项辅助能力。

29.3.4　框架内的事务管理支持

使用原始的JMS API进行消息的事务控制，是在创建Session的时候决定的，如下所示：

```
boolean sessionTransacted = true;
Connection connection = ...;
Session session = ➥
connection.createSession(sessionTransacted,Session.AUTO_ACKNOWLEDGE);
...
```

既然JmsTemplate已经对消息处理过程中涉及的API使用进行了封装，对于消息的事务控制自然也不能例外。实际上，Spring专门提供了针对JMS的PlatformTransactionManager实现，即JmsTransactionManager。这表明整个JMS相关的事务控制已经纳入了Spring的统一事务控制抽象层中，JmsTemplate和某些MessageListenerContainer的部分工作，就是使得自己能够加入到Spring统一事务抽象层这一大家族。

对于JMS相关的事务控制，我们可以有多种选择，如下所述。

- 如果事务只需要局限于消息处理的话，我们可以直接指定JmsTemplate或者相应MessageListenerContainer的“sessionTransacted”属性为true，来使用JMS本地的事务控制，

比如：

```
ConnectionFactory connectionFactory = ...;
JmsTemplate jmsTemplate = new JmsTemplate(connectionFactory);
jmsTemplate.setSessionTransacted(true);
...
```

- 在消息处理过程中，可能需要同时考虑消息以及数据访问相关的事务控制，这时，我们可以做进一步的划分，如下。
  - 如果消息的发送或者接收与数据访问之间不需要严格的事务界定，那么，可以让二者使用各自的事务管理器进行事务控制，即前者使用`JmsTransactionManager`，而后者使用`DataSourceTranactionManager`或者其他相关事务管理器。
  - 如果必须保证消息的发送或者接收与数据访问处于同一事务中，那么，我们就得使用`JtaTransactionManager`来管理全局性事务。不过，这需要我们使用的JMS Provider提供分布式的事务支持。

总地来说，在没有确实必要的情况下，应该尽量避免使用`JtaTranactionManager`开启全局事务控制。你需要根据具体场景来权衡各方案的复杂度与性能损耗之间的关系。

29.4 小结

消息机制在系统集成以及大型企业级应用中应用很广，可谓“不可或缺”之人才。JMS是Java平台为统一消息中间件产品的访问方式而提出的规范标准，但规范与应用之间毕竟存在差距。本章我们分析了JMS规范在应用中存在的一些问题或者说尴尬之处，然后引入Spring框架为简化基于JMS的应用开发所提供的各项“便利”，囊括了Spring框架为消息发送和同步消息接收、异步消息接收、JMS相关异常处理以及JMS相关场景下的事务处理等内容。希望本章的内容能够帮助你在以后的开发工作中更加高效、便捷地使用JMS！

第30章

使用Spring发送E-mail

本章内容

- 思甜前，先忆苦
- Spring的E-mail抽象层分析
- Spring的E-mail支持在实际开发中的应用

30.1 思甜前，先忆苦

E-mail本身的发展史可以说比互联网都要久远，其应用如此广泛，以至于我们可以在很多场合发现它的身影，这当然也包括我们日常所开发的各种应用程序。许多应用程序场景中都会用到E-mail，下面是常见的情景。

- 用户通过浏览器或者移动设备申请了某一系统账户，系统应用程序通常会将用户最终申请结果，以E-mail的方式发送给用户进行确认。
- 对于某些财务相关的系统来说，可能会定期地通过E-mail向系统中符合特定条件的用户发送指定的财务报表，或者出于宣传的考虑，如果用户订阅了某种服务，会定期通过E-mail的形式发送宣传资料，等等。
- 再有就是系统监控，如果应用程序出现问题，通常情况下，可以通过E-mail的形式通知系统管理员，等等。

Java平台上有许多E-mail的解决方案，而使用范围最广，名声也是最响的，自然非Sun的JavaMail（http://java.sun.com/products/javamail/）莫属了。JavaMail虽说强大，但API的使用比较烦琐。如果不能找到一种合适的方式来管理JavaMail的API的使用，即使发送一封简单的E-mail也并非易事。

鉴于Java平台E-mail解决方案的多样性，以及JavaMail在实际开发中不慎友好的表现，Spring提出了一套E-mail抽象层来简化应用程序中发送E-mail相关的开发工作。Spring的E-mail抽象层主要对JavaMail和COS（com.oreilly.servlet包提供的E-mail解决方案）这两种E-mail解决方案提供支持。不过，COS只支持简单文本邮件的发送，而且，Spring 2.5之后就取消了对其的支持，所以，余下的内容我们将主要以JavaMail为核心展开，虽然会在合适的位置顺带提及COS的相关信息。

在正式开始介绍Spring的E-mail抽象层之前，我们有必要先回头看一下，早先要使用JavaMail API发送一封E-mail是如何做的。代码清单30-1给出的代码恰好可以反映当时的情况。

代码清单30-1 通常情况下使用JavaMail原始API进行邮件发送的代码示例

```
String host = "your.mail.host";
String username = "..";
```

```
String password = "..";
String protocal = "smtp";

String mailSubject = "邮件标题";
String mailContext = "邮件内容";
String mailEncoding = "UTF-8";

boolean plainTextMail = false; // 是否发送普通文本文件

Properties props = new Properties();
props.put("mail.smtp.auth", "true");

Transport transport = null;
try
{
    Session session = Session.getDefaultInstance(props,null);
    MimeMessage message = new MimeMessage(session);
    message.setRecipients(RecipientType.TO, new Address[]{new InternetAddress➡
    ("toUser@gmail.com")});
    message.setRecipient(RecipientType.CC, new InternetAddress("toUser@gmail.com"));
    message.setFrom(new InternetAddress("fromUser@some.com"));
    message.setSubject(mailSubject, mailEncoding);

    if(plainTextMail)
    {
        message.setText(mailContext,mailEncoding);
    }
    else
    {
        Multipart multiPart = new MimeMultipart();
        BodyPart bodyPart = new MimeBodyPart();
        bodyPart.setText(mailContext);
        multiPart.addBodyPart(bodyPart);

        bodyPart = new MimeBodyPart();
        FileDataSource fds = new FileDataSource("attachment.name");// 添加附件
        bodyPart.setDataHandler(new DataHandler(fds));
        bodyPart.setFileName(fds.getName());
        multiPart.addBodyPart(bodyPart);

        message.setContent(multiPart);
    }
    transport = session.getTransport(protocal);
    transport.connect(host, username, password);
    transport.sendMessage(message, message.getAllRecipients());
}
catch(Exception e)
{
    // 根据情况处理异常
}
finally
{
    if(transport != null)
        try {
            transport.close();
        } catch (MessagingException e) {
            e.printStackTrace(); // 不要这样做
        }
}
```

30

虽然引入相应的变量声明使得这段代码看起来更长，但是，即使忽略最初的变量声明对这段代码的长度的贡献，我们依然可以发现，直接使用JavaMail API发送一封邮件真的不是什么省心的事情。

每次使用JavaMail API发送邮件，我们都需要设置JavaMail的Session需要的某些属性，然后根据Session构建相应的`MimeMessage`实例，并在构建完的`MimeMessage`实例上设置将要发送的消息的方方面面。在消息准备完成后，我们才能通过`Transport`连接到邮件服务器，并发送准备好的邮件信息。当然，最后还不要忘了异常的处理和`Transport`的关闭。而实际上，在这整个过程中，每次发送E-mail的时候我们都要关心的，可能只是要发送什么样的邮件内容，至于如何连接到邮件服务器之类的琐事，最好让它们离我们远远的，不是吗？所以，既然我们不想每次发送邮件都写这么一堆代码，自然就得考虑对整个过程进行适度地封装，以避免每次重复编码带给我们的烦恼。

JavaMail已经推出这么长的时间了，你在使用的过程中应该早就注意到了其使用上的烦琐，也许你早就已经实现了自己对JavaMail API的封装，也许直接使用了Jakarta Commons E-mail（http://commons.apache.org/email/）对JavaMail的封装实现。这里将介绍另一种选择，那就是Spring框架提供的E-mail抽象层，它让JavaMail的使用之路不再泥泞难行。

30.2　Spring 的 E-mail 抽象层分析

Spring所提供的E-mail抽象层主要体现在两个层次，即`MailMessage`抽象层次和`MailSender`抽象层次，我们可以从图30-1中获取一个初步的印象。

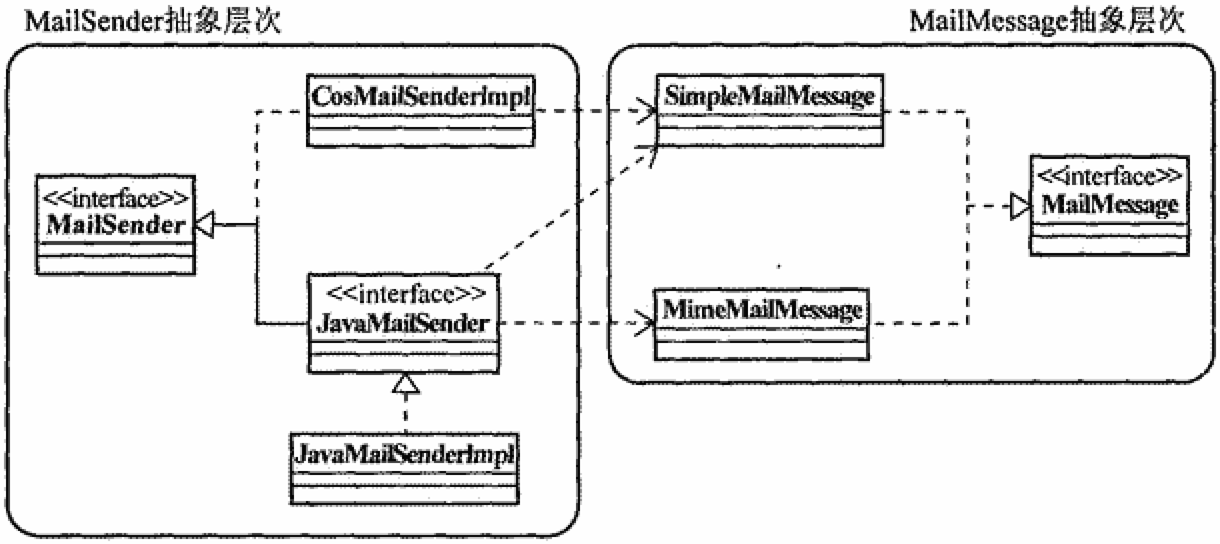

图30-1　Spring的E-mail抽象层设计示意图

`MailMessage`抽象层次旨在对要发送的邮件信息进行抽象和建模，`MailMessage`作为该层次的统领，定义了一套标准的邮件信息操作方式。如果说我们使用JavaMail API设置接收方地址之类的信息的时候，必须通过JavaMail API特定的`RecipientType`来指定接收方类型的话（如下）：

```
message.setRecipient(RecipientType.CC, new InternetAddress(".."));
```

那么，通过`MailMessage`，我们直接调用语义明确的setter方法来进行同样的操作即可，如下所示：

```
mailMessage.setTo(..);
mailMessage.setCc(..);
mailMessage.setBcc(..);
...
```

这种抽象最主要的好处就是，可以让我们不需要关心最终底层使用的是什么样的E-mail解决方案，完全以一种统一的方式来准备将要发生的邮件信息就可以了。

MailMessage主要有两个实现类，SimpleMailMessage代表普通的文本邮件（plain text mail），而MimeMailMessage则代表能够发送更丰富内容的MIME（Multipart Internet Message Encoding）类型的邮件信息。MimeMailMessage是构建在JavaMail的MimeMessage上的，除了可以像SimpleMailMessage那样指定普通的文本作为要发送的邮件内容，只要是MimeMessage能够添加到邮件的内容，对于MimeMailMessage来说都不在话下。HTML形式的邮件信息，添加了附件的邮件信息，都可以由MimeMailMessage统一构建并最终被发送。而说到MailMessage的发送，我们就得提到MailSender了。

有了要发送的邮件信息之后，剩下的当然就是找个人来发送它们，而这也正是MailSender抽象层次所要做的事情。MailSender抽象接口的主要目的就是屏蔽掉各种E-mail解决方案的差异性，不管底层的E-mail解决方案如何变换，我们只要把使用这些E-mail解决方案实现的MailSender实现类注入给依赖MailSender接口的对象即可，当前对象不需要知道具体的MailSender实现类是哪一个。MailSender的接口定义十分简单明了，如下所示：

```
public interface MailSender {
    void send(SimpleMailMessage simpleMessage) throws MailException;
    void send(SimpleMailMessage[] simpleMessages) throws MailException;
}
```

发送SimpleMailMessage的方法定义注定了直接的MailSender实现类只能发送普通的文本邮件，而COS类库恰好也只支持普通文本邮件，所以，CosMailSenderImpl就成为了第一个MailSender的直接实现类。如果我们现在要发送简单的普通文本邮件，那么现在就可以实现了，如代码清单30-2所示。

代码清单30-2 通过Spring的Mail抽象层发送简单文本文件的代码示例

```
// 1.准备要发送的邮件信息
SimpleMailMessage message = new SimpleMailMessage();
message.setTo("..");
message.setFrom("..");
message.setSubject("just a plain text mail");
message.setText("anything you like");
// 2.设置MailSender并发送邮件
MailSender mailSender = new CosMailSenderImpl();
setUpMailSender(mailSender); // mailSender.setHost("..");
mailSender.send(message);
```

因为CosMailSenderImpl的功能很简单，所以，没有办法指定邮件服务器所需要的必要的验证信息，而且，使用CosMailSenderImpl发送的SimpleMailMessage也不能指定replyTo和sendDate。大部分情况下，我们不太可能会考虑在生产环境下的应用程序中使用它，这也正是Spring 2.5之后就不再提供CosMailSenderImpl的原因了吧。

MailSender只能发送普通的文本邮件，而且其唯一的直接实现类CosMailSenderImpl在这方面的表现又是如此不尽如人意，看来，我们只好寻找更好的“继承人”了。

JavaMailSender扩展自MailSender，在MailSender发送普通文本文件功能基础上，添加了发送MIME邮件的功能。大多数情况下，我们使用JavaMailSender就行了，因为使用它发送什么形式的邮件都没有问题。从刚才的图30-1上我们也能发现，JavaMailSender发送的邮件形式指向了

SimpleMailMessage和MimeMailMessage两种类型。JavaMailSender唯一的一个实现类JavaMailSenderImpl使用JavaMail API实现，它完成了对JavaMail API的最终封装，可以说是最初使用JavaMail API发送邮件所经历的痛苦的终结者。至于JavaMailSenderImpl封装JavaMail API使用的方法，我不说你也应该猜出个大概，又是模板方法模式唱主角，代码清单30-3给出了该实现类相应方法的代码摘录。

代码清单30-3 JavaMailSenderImpl实现类相关模板方法代码摘录

```
public void doSend(MimeMessage[] messages2Send){
    // 预处理
    Transport transport = getTransport(getSession());
    transport.connect(getHost(), getPort(), getUsername(), getPassword());
    try {
        for (int i = 0; i < mimeMessages.length; i++) {
            MimeMessage mimeMessage = mimeMessages[i];
            transport.sendMessage(mimeMessage, mimeMessage.getAllRecipients());
        }
    }
    finally {
        transport.close();
    }
    // 后处理，比如异常的处理
}
```

JavaMailSender提供了两种常用的邮件发送方式，一种是直接外部创建要发送的MailMesssage实例，然后传给JavaMailSender发送；另一种则是使用JavaMailSender公开给我们的MimeMessagePreparator回调接口。下面让我们看一下使用这两种方式发送邮件具体是如何操作的。

30.2.1 直接创建邮件消息并发送

在发送邮件之前，我们先设定好要用于发送邮件的JavaMailSender，如以下代码所示：

```
JavaMailSender mailSender = new JavaMailSenderImpl();
JavaMailSenderImpl javaMailSender = (JavaMailSenderImpl)mailSender;
javaMailSender.setHost("..");
javaMailSender.setUsername("..");
javaMailSender.setPassword("..");

Properties javaMailProps = new Properties();
javaMailProps.put("mail.smtp.auth", "true");
javaMailSender.setJavaMailProperties(javaMailProps);
// mailSender现在可以使用
```

当然，所有这些都可以在Spring的IoC容器中以一个bean定义搞定。代码只是出于演示的目的，实际开发中，只需要声明JavaMailSender的依赖，然后将类似代码清单30-4所示的bean定义注入给相应的依赖对象即可。

代码清单30-4 JavaMailSenderImpl配置代码示例

```
<bean id="javaMailSender" class="org.springframework.mail.javamail.JavaMailSenderImpl">
    <property name="host" value=".."/>
    <property name="username" value=".."/>
    <property name="password" value=".."/>
    <property name="javaMailProperties">
        <props>
            <prop key="mail.smtp.auth">true</prop>
```

```
        </props>
    </property>
</bean>
```

提示 实例化一个可用的JavaMailSenderImpl可以有两种方式，第一种就是像我们这样，直接指定本地的一些配置属性，另一种方式就是直接指定JavaMail的Session。对于那些可以获取绑定到JNDI服务的Session的应用程序来说，后一种方式就是为其量身定做的。

因为JavaMail的MimeMessage不能独立于Session而创建，所以只有在有了可用的JavaMailSender之后，我们才能开始创建要发送的邮件信息。而这一目的要通过JavaMailSender的createMimeMessage()方法来达成，如下所示：

```
MimeMessage javaMailMessage = mailSender.createMimeMessage();
MimeMailMessage messageWrapper = new MimeMailMessage(javaMailMessage);
messageWrapper.setTo("..");
messageWrapper.setFrom("..");
messageWrapper.setSubject("..");
messageWrapper.setText("..");
// 其他设定……
mailSender.send(messageWrapper.getMimeMessage());
```

在准备要发送的邮件信息的时候，当然可以直接使用JavaMail的MimeMessage，但是，那样就不得不自己通过JavaMail的Multipart或者BodyPart来组装要发送的邮件内容。所以，我们使用了Spring提供的MimeMailMessage对createMimeMessage()返回的MimeMessage做简单的“包装”。现在，直接调用相应的setter方法设定邮件内容就可以了。在邮件准备完毕之后，最后当然就是直接调用JavaMailSender的send()方法发送，整个过程就是如此。

自己调用JavaMailSender的createMimeMessage()方法创建要发送的邮件实例当然没有什么大不了。不过，如果能够将创建邮件实例的工作从我们这里给砍掉的话，那岂不是更省事儿？这就是MimeMessagePreparator的工作。

30.2.2 使用 MimeMessagePreparator 发送邮件

JavaMailSender除了提供send(MimeMessage[]|MimeMessage)形式的邮件发送方法定义之外，还提供了使用MimeMessagePreparator回调接口的邮件发送方法。MimeMessagePreparator作为回调接口，可以获得JavaMailSender公开给它的MimeMessage实例。这样，我们只要在MimeMessage-Preparator内一心一意地准备邮件内容就可以了，至于这个MimeMessage是怎么创建的，由谁创建的，就无需去操心了。

依然沿用已经设定完毕的JavaMailSender实例，同样的邮件发送工作现在转而使用MimeMessagePreparator实现的话，那相应的实现代码看起来应该是如下的样子：

```
mailSender.send(new MimeMessagePreparator(){
    public void prepare(MimeMessage message) throws Exception {
        MimeMailMessage messageWrapper = new MimeMailMessage(message);
        messageWrapper.setTo("..");
        messageWrapper.setFrom("..");
        messageWrapper.setSubject("..");
        messageWrapper.setText("..");
    }
});
```

实际上，MimeMailMessage提供的封装支持比较弱。正如我们所看到的那样，我们只能设定一些

30

基本的信息，但是，要想以同样的setter方式来设定如邮件的附件等信息的话，MimeMailMessage就有些力不从心了。不过，总有后继者，MimeMessageHelper可以帮助我们完成更加丰富的邮件内容设定，代码清单30-5给了MimeMessageHelper一展身手的机会。

代码清单30-5 MimeMessageHelper使用代码示例

```
mailSender.send(new MimeMessagePreparator(){
    public void prepare(MimeMessage message) throws Exception {
    MimeMessageHelper messageHelper = new MimeMessageHelper(message, true, "UTF-8");
    messageHelper.setFrom("..");
    messageHelper.setTo("..");
    messageHelper.setSubject("..");
    StringBuilder builder = new StringBuilder("<html><head><meta http-equiv=➥
    'ContentType' content='text/html; charset=utf-8'></head><body><pre>");
    builder.append("<img src='cid:inlinePic'>");
    builder.append("</pre></body></html>");
    messageHelper.setText(builder.toString(), true);
    messageHelper.addInline("inlinePic", new ClassPathResource("img/picture.gif"));
    messageHelper.addAttachment("fileName.pdf", new ClassPathResource("yourFile.pdf"));
    }
});
```

使用MimeMessageHelper，让我们再也不用去操心Multipart以及BodyPart之间的关系，以及如何组合它们。现在，我们只要根据需要，以统一的setter方式添加必要的邮件内容就行了，不亦乐乎？

注意 JavaMailSender结合MimeMessagePreparator和MimeMessageHelper的组合，是使用JavaMailSender发送邮件的最常用方式。有关MimeMessageHelper的更多细节，不妨开发过程中参阅该类的Javadoc文档，其中有详细的功能设定说明。

30.3 Spring 的 E-mail 支持在实际开发中的应用

实际开发中，我们不可能像实例那样，直接通过字符串的拼接来创建所要发送的邮件的具体内容。更多时候，我们会使用系统指定的邮件模板。当发送给用户的邮件内容需要变更的时候（比如公司迁址，需要变更新的地址或者电话之类细节），我们不想挨个类地去改代码，而使用邮件模板的话，只要修改一下邮件模板就可以了。

从现在开始，我将向你展示如何在Spring中使用其E-mail支持来发送基于模板的电子邮件。虽然主要以Velocity作为使用的模板技术，但使用其他模板技术（比如FreeMarker）在道理上是相同的，只可能会存在少许具体操作上的差异。

在开始着手正式的工作之前，我们先构建一个要发送的邮件模板，使用Velocity的VTL编写模板如下：

```
来到魔法屋，就得听我的指示，我来选出符合条件的人，你来决定他们的命运！
嘿，轮到你了！ ${player}
```

以上邮件模板内容可以vm模板文件形式保存，也可以存入数据库中。这里先暂且将其存入magicHouse.vm文件，并将该文件放在Classpath的根路径下。

为了能够重用使用邮件模板发送邮件这一功能，我们对该功能进行抽象，抽象后的接口如代码清单30-6所示。

代码清单30-6 ITemplateMailAgent抽象接口定义

```
public interface ITemplateMailAgent {
    /**
     * convenient sending method with only recipient.TO <br/>
     */
    void sendMail(String receiver,String subject,String templateKey,Map<String,Object>➥
    context);
    /**
     * central method to send E-mail with template support.<br/>
     * @param receivers Recipients as a Parameter wrapper object.
     * @param templateKey the identity of the template to use.
     * @param context contains data that will be merged into template.
     */
    void sendMail(Recipients receivers,String subject,String templateKey,Map<String,➥
    Object> context);
}
```

当然，该抽象接口定义并不完善。如果要实际应用的话，或许要添加更多的方法来提供更为全面的支持，甚至，如果你觉得这种方法签名定义不爽，那就改成自己喜欢的风格。不管怎么样，我们将先就这一定义开始基于邮件模板的邮件发送整合之路。

ITemplateMailAgent的抽象允许我们提供基于不同模板技术的实现类，我们使用Velocity，所以，对应Velocity的ITemplateMailAgent实现类就此诞生了（见代码清单30-7）！

代码清单30-7 针对Velocity的ITemplateMailAgent实现类定义代码

```
public class VelocityTemplateMailAgent implements ITemplateMailAgent {
    private final String DEFAULT_SENDER = "sender@XXX.com.cn";
    private final String DEFAULT_ENCODING = "UTF-8";

    private String mailFrom = DEFAULT_SENDER;
    private String mailEncoding = DEFAULT_ENCODING;
    private JavaMailSender javaMailSender;
    private VelocityEngine velocityEngine;

    public void sendMail(String receiver, String subject,String templateKey, Map<String, ➥
    Object> context) {
        Recipients recipients = new Recipients(receiver);
        sendMail(recipients, subject, templateKey, context);
    }

    public void sendMail(final Recipients receivers,final String subject, String templateKey, ➥
    Map<String, Object> context)
    {
        validateRecipients(receivers);
        Validate.notEmpty(templateKey);
        StringWriter writer = new StringWriter();
        VelocityEngineUtils.mergeTemplate(velocityEngine, templateKey, ➥
        getMailEncoding(),context, writer);
        final String mailText = writer.toString(); // mail content is ready

        getJavaMailSender().send(new MimeMessagePreparator(){
            public void prepare(MimeMessage message) throws Exception {
            MimeMessageHelper helper = new MimeMessageHelper(message,getMailEncoding());
            helper.setFrom(getMailFrom());
            helper.setTo(receivers.getTo());
            if(!CollectionUtils.isEmpty(receivers.getCcList()))
                helper.setCc(receivers.getCc());
```

30

```
            if(!CollectionUtils.isEmpty(receivers.getBccList()))
                helper.setBcc(receivers.getBcc());
                helper.setSubject(subject);
                helper.setText(mailText);
        }
      });
    }

    private void validateRecipients(Recipients receivers) {
        //...
    }
    // 用于依赖注入的getter和setter方法定义
}
```

VelocityTemplateMailAgent的实现逻辑依赖于两个主要组件，这可以推而广之到使用其他模板技术的ITemplateMailAgent实现类中，如下所述。

- 模板引擎——VelocityEngine。我们需要使用具体的模板引擎来合并具体的邮件模板和要发送的数据，合并后的结果自然就成为我们将要最终发送的邮件内容。
- JavaMailSender。没有JavaMailSender的支持，我们好像也发送不了邮件吧？（当然，使用其他E-mail封装方案则另当别论。）

至于其他的实现细节，我想从代码上来看，并不是太难理解，所以，这里就不做太多解释了。

> **注意**　VelocityTemplateMailAgent的当前实现只支持基于邮件模板的普通文本邮件的发送。如果要支持MIME邮件，那需要做更多的工作，你不妨自己尝试一下。

现在只要将VelocityTemplateMailAgent和它的相关依赖添加到Spring的IoC容器，或者编程组装在一起，我们就能发送基于邮件模板的邮件了。当前，我们不妨像通常那样，通过Spring的IoC容器来做这部分工作，如代码清单30-8所示。

代码清单30-8　VelocityTemplateMailAgent及其相关实现类使用配置示例

```
<bean id="cpLoaderVelocityEngine" ➥
class="org.springframework.ui.velocity.VelocityEngineFactoryBean">
    <property name="configLocation"➥
    value="classpath:cn/spring21/conf/velocity-config. ➥
    properties"/>
</bean>

<bean id="javaMailSender"➥
class="org.springframework.mail.javamail.JavaMailSenderImpl">
    <property name="host" value=".."/>
    <property name="username" value=".."/>
    <property name="password" value=".."/>
    <property name="javaMailProperties">
        <props>
            <prop key="mail.smtp.auth">true</prop>
        </props>
    </property>
</bean>

<bean id="templatEmailAgent" class="org.darrenstudio.books.unveilspring.mail.template.
```

```
VelocityTemplateMailAgent">
    <property name="javaMailSender" ref="javaMailSender"></property>
    <property name="velocityEngine" ref="cpLoaderVelocityEngine"/>
</bean>
```

VelocityTemplateMailAgent所依赖的VelocityEngine是我们通过Spring提供的VelocityEngineFactoryBean注入给它的。VelocityEngineFactoryBean作为一个FactoryBean实现，封装了一个VelocityEngine相关的设定，使我们可以简单bean的形式将VelocityEngine添加到Spring的IoC容器进行管理。在Spring MVC部分，我们也应该接触过该类了。VelocityEngineFactoryBean使用的配置文件内容比较简单，如下所示，当然，如果需要，你可以在使用的时候追加更多控制项：

```
# velocity-config.properties
resource.loader = classpath
classpath.resource.loader.description = Classpath Resource Loader
classpath.resource.loader.class = org.apache.velocity.runtime.resource.loader. ➥
ClasspathResourceLoader

velocimacro.library =
```

我们的velocity-config.properties配置，使得VelocityEngine将从Classpath加载vm模板文件。所以，现在我们可以通过如下的形式发送最初模板所设定的邮件内容了：

```
VelocityTemplateMailAgent mailAgent = ...// 通过IoC容器注入或查找相应实例
Map<String,Object> context = new HashMap<String, Object>();
context.put("player", "孙小美");
mailAgent.sendMail("fujohnwang@gmail.com", "邮件标题", "magicHouse.vm", context);
```

像收件人、邮件标题，以及合并到邮件模板的数据，通常需要我们根据应用程序的上下文来获得。不管怎么样，如果你想看一下效果的话，那图30-2是我的Gmail所呈现的。

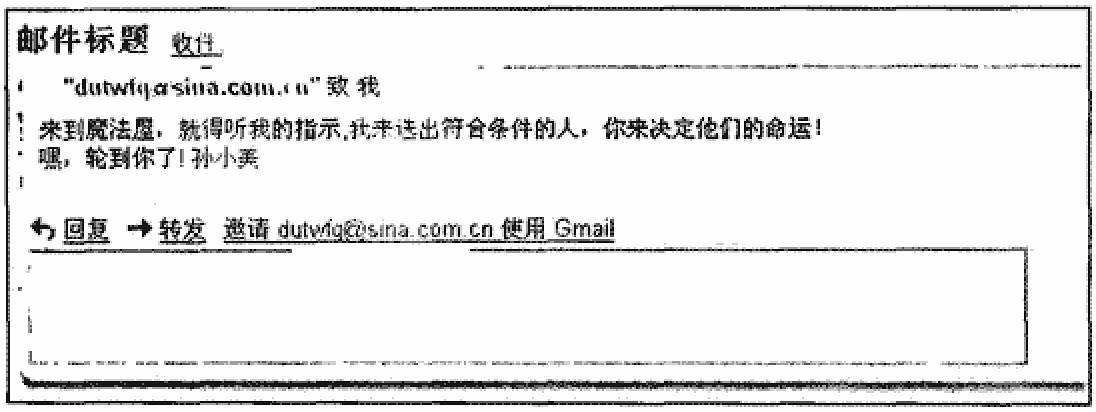

图30-2 邮件接收效果演示图

看起来已经达到预期的效果了！

> **注意** 有关结合Velocity模板和Spring发送邮件的内容，最新的Spring文档中都有介绍。不过，如果我没有记错的话，最早提出这种实践思路的是一篇文章“Sending Velocity-Based E-Mail With Spring”。如果你感兴趣的话，不妨根据本书后面的参考资料找这篇文章读读。

将模板文件放入应用程序的Classpath或者直接放到文件系统中，有时候会因为应用程序场景的变化产生些许不便。比如，多个应用程序都需要访问同一模板内容的话，这些应用程序可能需要将同一

份模板分别打包，或者通过其他途径，来访问文件系统中存放的同一份模板文件。所以，出于这些场景的考虑，我们的模板文件也会被放入某种类型的数据存储服务中，比如数据库或者目录服务器上。这种情况下，我们就需要在使用VelocityTemplateMailAgent之前，做点儿附加工作，以便让VelocityEngine可以加载位于特定数据存储服务中的邮件模板。

假设我们将magicHouse.vm以及系统中其他邮件模板的内容存入数据库。为了让Velocity能够从数据库加载这些模板资源，我们可以替换Velocity最初使用的ResourceLoader实现，从ClasspathResourceLoader转而使用DataSourceResourceLoader。整个切换过程涉及的，实际上也只是velocity-config.properties配置内容的少许改变。应该说，这种思路是正确的。不过，实施当中我们可能会碰到点儿小问题！

通常，使用DataSourceResourceLoader的配置文件内容可能是如代码清单30-9所示的样子。

代码清单30-9 Velocity使用DataSourceResourceLoader的可能配置代码示例

```
resource.loader = ds
ds.resource.loader.public.name = DataSource
ds.resource.loader.description = Velocity DataSource Resource Loader
ds.resource.loader.class = org.apache.velocity.runtime.resource.loader. ➥
DataSourceResourceLoader
ds.resource.loader.resource.datasource = java:comp/env/jdbc/Velocity
ds.resource.loader.resource.table = mail_templates
ds.resource.loader.resource.keycolumn = TEMPLATE_KEY
ds.resource.loader.resource.templatecolumn = TEMPLATE_DEFINIATION
ds.resource.loader.resource.timestampcolumn = UPDATE_DATE
ds.resource.loader.cache = false
ds.resource.loader.modificationCheckInterval = 60
```

从中可以看出，DataSourceResourceLoader只能使用从JNDI获取的DataSource来加载模板资源。如果要使用外部独立的数据源，那么需要做一些类似如下的编码工作：

```
DataSourceResourceLoader ds = new DataSourceResourceLoader();
DataSource dataSource = ...; // 可以从其他地方注入
ds.setDataSource(dataSource);
velocityEngine.setProperty("ds.resource.loader.instance",ds);
```

如果Spring的VelocityEngineFactoryBean提供了setResourceLoader(..)的方法的话(实际上该名称的方法确实提供了，但却是接受Spring的ResourceLoader类型作为方法参数，而不是Velocity的ResourceLoader类型)，我们可以通过一个FactoryBean来封装以上代码所示的对DataSourceResourceLoader进行设定的相关逻辑，但不巧，事情没有按照我们所预想的方向发展，我们得另寻他路。

VelocityEngineFactoryBean定义有一个postProcessVelocityEngine(VelocityEngine)方法。该方法为protected，可以允许子类覆写它，以对VelocityEngineFactoryBean所管理的Velocity-Engine实例做进一步的定制。所以，我们可以从这里着手，解决DataSourceResourceLoader使用外部独立数据源的问题。既然VelocityEngineFactoryBean没有提供我们所需要的设置选项，那我们就扩展它。扩展后的VelocityEngineFactoryBean如代码清单30-10所示。

代码清单30-10 VelocityEngineFactoryBean扩展类实现代码示例

```
public class ExtendedVelocityEngineFactoryBean extends VelocityEngineFactoryBean
{
    private DataSource dataSource;
```

```
    @Override
    protected void postProcessVelocityEngine(VelocityEngine velocityEngine) throws➥
    IOException, VelocityException
    {
        super.postProcessVelocityEngine(velocityEngine);
        DataSourceResourceLoader resourceLoader = new DataSourceResourceLoader();
        resourceLoader.setDataSource(dataSource);
        velocityEngine.setProperty("ds.resource.loader.instance", resourceLoader);
    }

    public DataSource getDataSource() {
        return dataSource;
    }

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }
}
```

现在，以ExtendedVelocityEngineFactoryBean替换掉最初的VelocityEngineFactoryBean，并注入给VelocityTemplateMailAgent，我们的VelocityTemplateMailAgent就可以从数据库来加载邮件模板了。

不过好事多磨，在此之前，需要注意余下的几个问题。

(1) 启用了使用外部数据源的DataSourceResourceLoader之后，velocity-config.properties中有关内容需要注释掉（或者直接删掉），否则，Velocity依然会沿用配置文件中的配置内容。注释后的配置文件如代码清单30-11所示。

代码清单30-11 注释后的Velocity配置文件示例

```
...
ds.resource.loader.public.name = DataSource
ds.resource.loader.description = Velocity DataSource Resource Loader
# ds.resource.loader.class = org.apache.velocity.runtime.resource.loader. ➥
DataSourceResourceLoader
# ds.resource.loader.resource.datasource = java:comp/env/jdbc/Velocity
ds.resource.loader.resource.table = mail_templates
ds.resource.loader.resource.keycolumn = TEMPLATE_KEY
ds.resource.loader.resource.templatecolumn = TEMPLATE_DEFINIATION
ds.resource.loader.resource.timestampcolumn = UPDATE_DATE
...
```

(2) 说是从数据库加载模板，我们还没有将模板内容导入数据库，所以，根据配置内容所示，我们需要建立一个名为mail_templates的表，然后声明模板对应的标志列和模板的定义内容列等，表定义DDL如下所示：

```
CREATE TABLE mail_templates (
    TEMPLATE_KEY             varchar(25) NOT NULL,
    TEMPLATE_DEFINIATION     text NOT NULL,
    UPDATE_DATE              datetime NOT NULL,
    CREATED_DATE             datetime NOT NULL,
    PRIMARY KEY(TEMPLATE_KEY)
)
```

至于实际系统中，要以什么作为表名，以什么列唯一标志模板，以及以什么列来保存模板内容，那是你自己的事情了，无非就是更改一下velocity-config.properties的配置内容而已。

假如我们通过如下SQL插入模板内容的话：

```
insert into mail_templates values('rich4','来到魔法屋，就得听我的指示,我来选出符合条件的人，你来决定他们的命运！\n嘿，轮到你了！ ${player}',NOW(),NOW())
```

那使用VelocityTemplateMailAgent进行邮件发送的如下代码，可以获得与从Classpath或者文件系统加载邮件模板同样的效果：

```
VelocityTemplateMailAgent mailAgent = // 通过Ioc容器注入或者查找相应实例;
Map<String,Object> context = new HashMap<String, Object>();
context.put("player", "孙小美");
mailAgent.sendMail("fujohnwang@gmail.com", "邮件标题", "rich4", context);
```

其实只是模板文件名换成了数据库表中的标志键。

> **提示**　以上几种场景都是基于Velocity进行介绍的，如果你愿意使用其他模板技术，可以在此基础上进行适度的调整，也可以达到同样的目的。实际上，不使用Velocity的DataSource-ResourceLoader，我们同样可以达到从数据库获取邮件模板的目的，而且，还要更加灵活，适用范围也更广，能猜到是什么吗？（想想Spring的数据访问相关内容。）

30.4　小结

与Java EE平台上许多API的境遇类似，JavaMail API虽强大，但依然没能摆脱实际应用中的烦琐。本章我们先忆苦，简单回顾了早些时候JavaMail API的实践之路，在此基础上，我们引入Spring的Mail抽象层相关内容。我们一起了解了Spring的Mail抽象层的方方面面，同时也对Spring的Mail抽象层在实际开发用的应用场景进行了简单的探索。希望你结束本章的阅读之后，能够借助Spring的Mail抽象层的支持，“多快好省”地完成日常开发中各种Mail相关功能。

第31章

Spring中的任务调度和线程池支持

本章内容

- Spring与Quartz
- Spring对JDK Timer的集成
- Executor的孪生兄弟TaskExecutor

在我们眼中，或许批处理（Batch Processing）是那么的不引人注目，但不可否认的一点是，几乎所有的企业级应用程序都会或多或少地依赖于相关的批处理程序。可以说，批处理在企业级应用程序中占据重要地位，这或许也正是Spring框架的娘家Interface21（现在叫SpringSource）要特地发起Spring Batch（http://static.springframework.org/spring-batch/）子项目的原因吧！

为了能够有效地管理批处理程序中的各种批处理任务（Batch Job），我们通常会根据当前场景需求，选用某种合适的任务调度程序（Job Scheduler）。至于可供我们选择的调度程序，则可能随需求的不同而变化，从操作系统自身提供的简单任务调度功能支持（比如，Unix/Linux的cron，或者Windows的“计划任务”），到开源的调度程序实现（比如OpenSymphony的Quartz（http://www.opensymphony.com/quartz/）），一直到功能更为强劲的商业产品（比如BMC的CONTROL-M或者CA的AutoSys），可以说是不一而足。虽然商业产品可以提供更多更强劲的功能支持，比如灾难恢复、运行期间任务状态监控等，但在需求不是很复杂的情况下，使用商业产品显得有些“大材小用”的感觉。对于一般的任务调度需求，普通的任务调度程序就能满足要求，比如Open Symphony的Quartz。我们不打算就批处理的历史，甚至所有的任务调度产品的秉性做什么长篇大论，JDK 1.3之后引入的Timer和OpenSymphone的Quartz是我们即将开幕的这场戏的主角，尤其是在有Spring做它们后盾的情况下。

31.1 Spring 与 Quartz

31.1.1 初识 Quartz

Quartz是Open Symphony（http://www.opensymphony.com/）[①]开发的一款开源的任务调度框架（或者说任务调度引擎）。相对于JDK的Timer之类的简单任务调度程序来说，Quartz拥有更为全面的功能支持： 31

- 允许批处理任务状态的持久化，并且提供不同的持久化策略支持；

① Open Symphony开发有多款成功的开源产品，包括著名的Web框架WebWork、OSCache缓存解决方案、OSWorkflow工作流产品等。

- 批处理任务的远程调度；
- 提供基于Web的监控接口；
- 集群支持；
- 插件式的可扩展性。

要想完全理解和使用Quartz所有的功能特性，或许需要些许时日。不过，如果想快速上手使用Quartz进行任务调度的话，仅理解Quartz相关的几个基本概念，或许真的使得这一想法变得可行。Quartz拥有明确的角色划分，分别对应处理任务调度过程中的各种相关关注点，如下所述。

- Job。代表将被调度的任务，一个任务要想让Quartz能够调度它，Job的帽子是必须戴上的。
- JobDetail。JobDetail的主要职能是提供Job执行的上下文信息，Job所依赖的某些配置信息，可以通过JobDetail提供，二者通过JobDataMap进行数据交互。
- Trigger。Trigger用于触发被调度任务的执行，可以根据不同的触发条件，提供不同的Trigger实现。
- Scheduler。Quartz的核心调度程序，负责管理所有相关的Trigger和JobDetail，是最终掌管调度任务生杀大权的“人”。

图31-1或许可以更为形象地描述Quartz各种角色之间的关系。

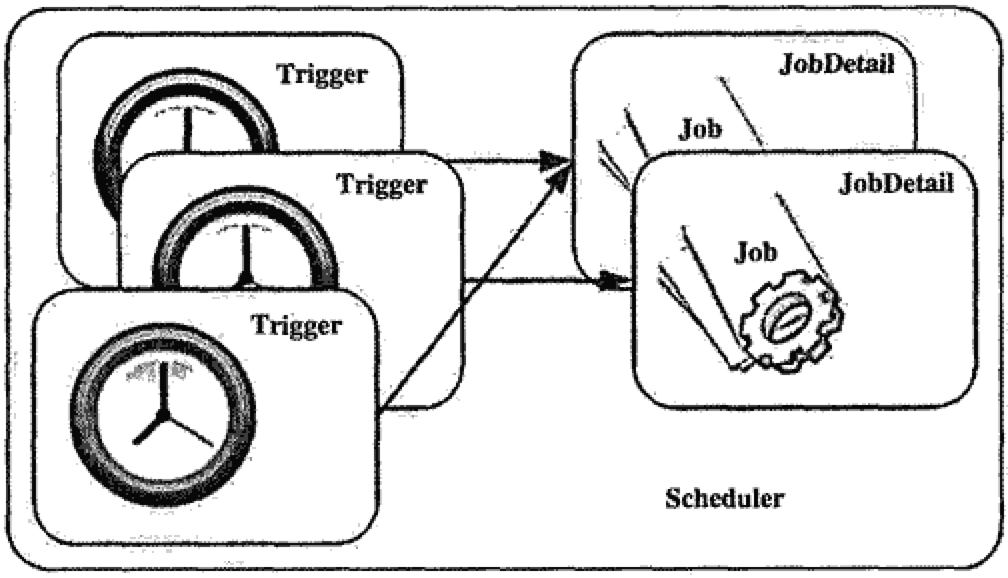

图31-1　Quartz相关概念/角色关系示意图

通常情况下，一个Trigger只能用于一个Job，但多个Trigger可以用于同一Job。这一特性可以帮助我们实现更加复杂的调度需求，稍后我们将就这一点做更多阐述。

要使用Quartz进行任务调度，首先我们必须有能够被调度的任务实现，给将被调度的对象冠以Job或者StatefulJob的“头衔”，可以让我们得到一个可被Quartz调度的任务类。还记得我们最初的FXNewProvider吗？现在该是让它真正上场的时间了！现在，我们构建FXNewsJob，以便Quartz可以定时调度它去抓取相应的外汇新闻信息，具体实现如代码清单31-1所示。

代码清单31-1　FXNewsJob定义代码

```
public class FXNewsJob implements Job {
    private FXNewsProvider fxNewsProvider;

    public void execute(JobExecutionContext jobExecCtx) throws JobExecutionException {
        getFxNewsProvider().getAndPersistNews();
```

```
    }

    public FXNewsProvider getFxNewsProvider() {
        return fxNewsProvider;
    }

    public void setFxNewsProvider(FXNewsProvider fxNewsProvider) {
        this.fxNewsProvider = fxNewsProvider;
    }
}
```

有了要调度的任务，接着需要考虑的问题就是，应该在什么样的时机下调用它，而这也正是Trigger存在的意义。Quartz中最经常使用的且是主要的两种Trigger实现类，一个是SimpleTrigger，可以指定简单的基于时间间隔的调度规则；另一个就是CronTrigger，可以类似于Unix/Linux操作系统中Cron程序所使用的表达式来指定调度规则。通常情况下，仅使用CronTrigger即可满足所有的调度需求。在Quartz中，我们可以直接实例化相应的Trigger实现类并使用，例如：

```
Trigger simpleTrigger = new SimpleTrigger("triggerName",Scheduler.DEFAULT_GROUP,new ➥
Date(),null,SimpleTrigger.REPEAT_INDEFINITELY,60000);

Trigger cronTrigger = new CronTrigger("cronTriggerName",Scheduler.DEFAULT_GROUP, ➥
"0 0/1 * * * ?");
```

以上两个Trigger定义完成同样的功能，将使得对应的调度任务每隔一分钟被调度一次。Quartz中的每一Trigger都归属于相应的组，在它们各自的组中需要有唯一的名称。在我们的定义中，我们让simpleTrigger和cronTrigger同样归属于Scheduler的默认组。有关两种Trigger的更多信息，请参照对应的Javadoc文档，我们就不再做更多解释了，尤其是CronTrigger，要发挥它的最大能量，足够熟悉cron表达式是必须的。

在Job和Trigger都具备之后，我们就该通过Scheduler将它们关联起来并进行最终的任务调度，如下所示：

```
Scheduler scheduler = new  StdSchedulerFactory().getScheduler();
scheduler.start();

JobDetail jobDetail = new JobDetail("jobName",Scheduler.DEFAULT_GROUP,FXNewsJob.class);
Trigger cronTrigger = new CronTrigger("cronTriggerName",Scheduler.DEFAULT_GROUP, ➥
"0 0/1 * * * ?");

scheduler.scheduleJob(jobDetail, cronTrigger);
```

Scheduler是从相应的Factory中获取的。在使用获取到的Scheduler进行任务调度之前，需要start()该Scheduler。我们可以发现，相应的Job不是由我们直接实例化，而是通过JobDetail以Class类型提供的，也就是说，Scheduler将通过反射来实例化并执行相应的调度任务。在start()之后再调用Scheduler的scheduleJob(..)方法，我们的FXNewsJob就算被提交执行了。整个过程看起来好像并不像想象的那么难，不是吗？

31

> **注意** 有关Quartz框架的更多信息可以参考Open Symphony官方网站提供的详细文档。另外，*Quartz Job Scheduling Framework*一书对Quartz框架给予了全面的介绍，阅读此书是了解Quartz的最佳途径。

原理上来说，使用Quartz进行任务调度的基本过程就是这个样子。不过，你觉得我们的FXNewsJob

可以被正确地调度吗？实际上，我根本就没有去运行这段调度程序，因为我知道它注定执行不了，它所依赖的所有相关对象都没有实例化，它又怎么会成功运行呢？为了让被调度的任务能够享受依赖注入等一系列服务，也为了让Quartz的使用更加便捷和可配置，Spring对Quartz的集成需求自然也就该浮出水面了。

31.1.2 融入 Spring 大家庭的 Quartz

Spring对Quartz的集成主要体现在使用的便利性上，比如，为各种Quartz相关的概念实体（Job、JobDetail等）提供合理的默认值，或者提供更加接近普通bean风格的配置形式等。当然，最主要的，纳入Spring框架管理的Quartz，将很自然地获得依赖注入和AOP等相关服务。

在对Quartz（甚至Timer）的集成过程中，Spring基本上是清一色地采用了FactoryBean这一“制式装备”。只要抓住这一特点，整个Spring对Quartz的集成就变得容易理解多了。

1. Job的实现策略

在Quartz中，每一Job所需要的执行上下文（Execution Context）信息是由其对应的JobDetail提供的，二者通过JobDataMap进行数据通信。如果我们通过JobDetail的JobDataMap设置Job执行的上下文信息，如下所示：

```
JobDetail jobDetail = new JobDetail("jobName",Scheduler.DEFAULT_GROUP,HelloWorldJob. ➥
class);
jobDetail.getJobDataMap().put("message", "helloWorld");
jobDetail.getJobDataMap().put("counter", 10);
```

那么，在对应的HelloWorldJob中，我们就可以通过如下方式获得这些上下文信息，并使用它们：

```
public class HelloWorldJob implements Job {
public void execute(JobExecutionContext ctx) throws JobExecutionException {
    JobDataMap jobDataMap = ctx.getJobDetail().getJobDataMap();
    String messsage = jobDataMap.getString("message");
    int    counter = jobDataMap.getInt("counter");
    // ...
    }
}
```

为了改善具体Job实现类获取上下文信息的“生态环境”，Spring提供了org.springframework.scheduling.quartz.QuartzJobBean。在实现Job的时候，通过继承QuartzJobBean，而不是直接实现Job接口，可以让我们在Job实现类中，直接以bean属性的形式访问当前Job执行的上下文信息。代码清单31-2给出了一个的QuartzJobBean实现实例。

代码清单31-2 QuartzJobBean实现代码示例

```
public class HelloWorldJobExtendingQuartzJobBean extends QuartzJobBean {
    private String message;
    private int    counter;
    @Override
    protected void executeInternal(JobExecutionContext arg0) ➥
    throws JobExecutionException {
        // getMessage() 并使用
        // getCounter() 并使用
    }
    public String getMessage() {
        return message;
    }
    public void setMessage(String message) {
```

```
        this.message = message;
    }
    public int getCounter() {
        return counter;
    }
    public void setCounter(int counter) {
        this.counter = counter;
    }
}
```

QuartzJobBean所做的工作，将保证上下文信息在executeInternal(JobExecutionContext)方法执行之前被逐一注入到相应的bean属性中。如果只从Job执行上下文信息的获取这一点来看，QuartzJobBean算是一种进步，但实际上，QuartzJobBean本身的极具侵入性，却使得它基本丧失存在的价值。如果某一Job实现确实需要在执行之前做一些后处理的话，比如刚才的获取执行上下文信息，那么完全可以提供相应的后处理机制来实现，而没有必要非得强制人家去继承某一父类，不是吗？不过，这或许是Quartz历史遗留问题导致的，在1.5或者更高版本之后，Quartz提供了一种JobFactory机制，允许对要执行的Job进行定制。这样，相应的Job实现就不需要受到任何父类继承的限制了。Spring提供的org.springframework.scheduling.quartz.SpringBeanJobFactory就是一种能够完成QuartzJobBean同样使命的JobFactory实现。所以，如果你能够使用Quartz 1.5或者更高版本，那么把QuartzJobBean完全放到脑后吧！

2. JobDetail的更多选择

按照Quartz的惯例，一个典型的Job实现类，最终需要一个对应的JobDetail来帮助提供执行的上下文信息。现在，我们可以通过org.springframework.scheduling.quartz. JobDetailBean来创建和配置相应Job所对应的JobDetail实例，如代码清单31-3所示。

代码清单31-3 JobDetailBean配置代码示例

```
<bean id="fxNewsJobDetail" class="org.springframework.scheduling.quartz.JobDetailBean">
    <property name="jobClass" value="HelloWorldJob | HelloWorldJobExtendingQuartzJobBean"/>
    <property name="jobDataAsMap">
        <map>
            <entry key="message">
                <value>HelloWorld</value>
            </entry>
            <entry key="">
                <value>10</value>
            </entry>
        </map>
    </property>
</bean>
```

可以看到，原先通过编码指定的Job类型和添加到JobDataMap的执行上下文信息，现在全部可以通过配置的方式完成设置。而且，原先编码实例化JobDetail的时候，必须指定JobDetail所在的组和组内的唯一标志名称，现在即使我们不明确指定，JobDetailBean也会提供合理的默认值（以bean定义的名称作为JobDetail的名称，使用DEFAULT组作为jobDetail的组）。

31

JobDetailBean支持Spring为JobDetail提供的最基础的集成设施。不过，MethodInvoking-JobDetailFactoryBean或许更加讨人喜爱一些。对于将被调度的业务逻辑来说，合理情况下，应该以独立的形式而存在，不应该因为某些业务逻辑可能会被调度执行，就将它们直接编码到Job实现类

中（或者Timer的TimerTask中）。这样，当需要对它们进行调度的时候，只需要根据当前情况提供相应的Job实现类，让Job实现类在调度方法中调用相应的业务对象即可，而我们的FXNewsProvider和FXNewsJob之间的关系，实际上就是如此。不过，要是调度程序执行的业务逻辑涉及多个独立的业务对象，我们可能就得为每个业务对象都提供一个薄薄的Job实现类了，而实际上，这些Job实现类中可能也只是调用业务对象方法的一行代码而已，MethodInvokingJobDetailFactoryBean的出现可以让我们避免这种尴尬。

实际上，对于FXNewsProvider的执行调度来说，我们也确实没有提供单独的FXNewsJob实现，而是直接让MethodInvokingJobDetailFactoryBean帮我们在执行的时候直接调用FXNewsProvider的相应方法，如下配置代码所示：

```
<bean id="jobDetail" ➥
class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">
    <property name="targetObject" ref="fxNewsProvider"/>
    <property name="targetMethod" value="getAndPersistNews"/>
</bean>
```

使用MethodInvokingJobDetailFactoryBean，我们只需要指定调度执行时候应该调用哪个对象实例的哪一方法就可以了，MethodInvokingJobDetailFactoryBean将根据这些信息，通过反射完成指定业务方法的调用。至于MethodInvokingJobDetailFactoryBean作为JobDetail为什么能够接受非Job类型的对象作为Job（FXNewsProvider之类的对象并不需要实现Job之类的接口），原理上实际也很简单，MethodInvokingJobDetailFactoryBean会在内部构建相应的Job实现类（MethodInvokingJobDetailFactoryBean.MethodInvokingJob 和 MethodInvokingJobDetailFactoryBean.StatefulMethodInvokingJob），这些Job实现类会在合适的时机调用指定给MethodInvokingJobDetailFactoryBean的业务对象上的业务方法。所以，你看，MethodInvokingJobDetailFactoryBean实际上并没有“推卸”它作为JobDetail必须管理Job类型调度任务的责任。

虽然使用MethodInvokingJobDetailFactoryBean可以带给我们很大的便利，但是，通过它所返回的JobDetail信息是不可序列化的，从而也就无法保存或者说持久化。所以，对于状态紧要的调度任务来说，还是应该给出具体的Job实现类，并结合JobDetailBean进行管理。

> **注意** 有关JobDetailBean和MethodInvokingJobDetailFactoryBean的更多配置项，在实际开发过程中，请直接参考相应类的Javadoc文档。

3. Trigger的可配置化

Quartz的两种Trigger实现SimpleTrigger和CronTrigger，实际上已经可以作为普通的JavaBean添加到Spring的IoC容器进行管理，但是，在实例化的时候，它们都需要指定相应的管理组和组内唯一标志名称。考虑到这一点，Spring分别为它们提供了对应的SimpleTriggerBean和CronTriggerBean封装类。SimpleTriggerBean和CronTriggerBean可以提供合理的默认值，比如以bean定义名称直接作为Trigger名称，以DEFAULT组作为trigger默认的组等，从而免去每次都要指定所在组和组内标志名称的琐事。当然，SimpleTriggerBean和CronTriggerBean也同样是采用FactoryBean机制实现的。

基本上，原先SimpleTrigger和CronTrigger所拥有的属性，在配置SimpleTriggerBean和CronTriggerBean的时候都可以使用，代码清单31-4给出了一段二者的配置代码示例。

代码清单31-4 SimpleTriggerBean和CronTriggerBean配置代码示例

```
<bean id="simpleTrigger" class="org.springframework.scheduling.quartz.SimpleTriggerBean">
    <property name="jobDetail" ref="jobDetail"/>
    <property name="repeatInterval" value="3000"/>
</bean>

<bean id="cronTrigger" class="org.springframework.scheduling.quartz.CronTriggerBean">
    <property name="jobDetail" ref="jobDetail"/>
    <property name="cronExpression" value="0 0/1 * * * ?"/>
</bean>
```

对于二者来说，共同的配置项就是jobDetail。在最初介绍Quartz的时候我们提过，每个Trigger都只对应一个Job（或者说JobDetail，因为Job和JobDetail通常是一对一的），所以，为了让Quartz的Scheduler在调度的时候，可以直接以scheduler.scheduleJob(trigger);的形式，而不是最初的scheduler.scheduleJob(jobDetail, cronTrigger);形式进行调度， 我们通常需要在配置Trigger的时候将与之关联的JobDetail指定给它。

在实际开发过程中，通常会遇到比较棘手的调度需求。对于这种情况，使用单一的Trigger往往达不到效果，即使是采用表达能力较强的CronTrigger也是如此，这时，我们不妨从以下两种方向进行考虑。

- 虽然一个Trigger同时只能对应一个Job，但是多个Trigger所对应的Job却可以是同一个， 所以，我们可以通过组合多个Trigger的形式来指定同一Job的调度规则，一个Trigger完成总调度规则的一个规则子集，最终所有Trigger的规则自己取并集就是最终我们所需要的调度规则，如图31-2所示。

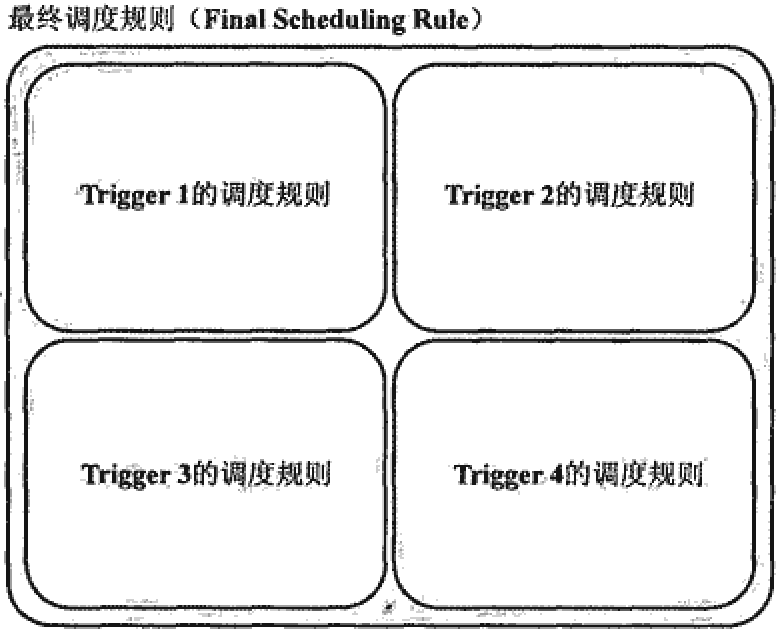

图 31-2

- 多个Trigger规则的附加或许更多的是为了补足单一Trigger调度规则覆盖范围不够的问题。如果单一Trigger的规则覆盖范围足够大，只是覆盖范围内有小部分调度规则需要剔除，我们可以结合Trigger与Quartz提供的Calender来达成最终所需要的调度规则。Trigger负责提供覆盖范围足够广的调度规则，而Calender负责排除这一规则范围内不需要的部分，如图31-3所示。

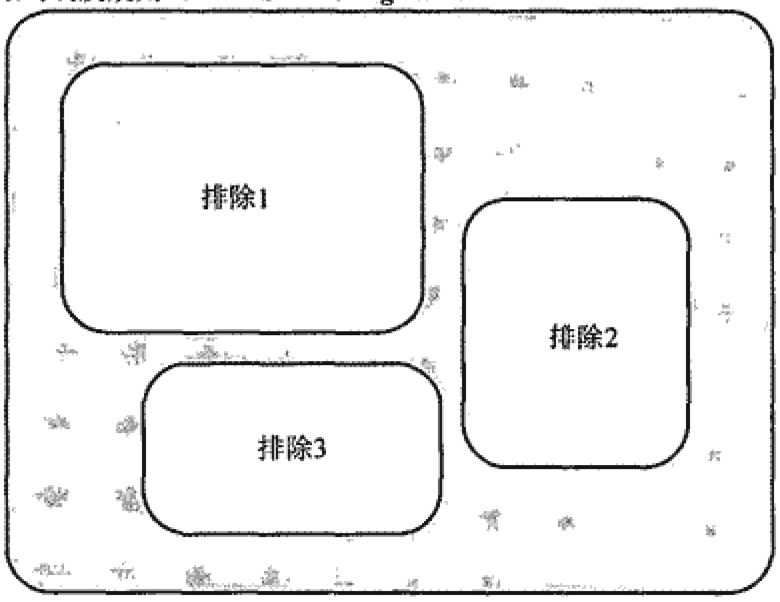

图　31-3

有关Calender和HolidayCalender的信息，可以参考Quartz相关文档。

当然，解决问题的方式肯定不止一种两种，如果你可以想出更好的解决方式，一定得告诉我。

4. Scheduler的新家

Scheduler本来就是从相应的Factory中来的（Scheduler scheduler = new StdSchedulerFactory().getScheduler()），所以，使用FactoryBean对其进行封装看起来也是很自然的事情。Spring提供了org.springframework.scheduling.quartz.SchedulerFactoryBean对Quartz的Scheduler进行管理，通过它，我们可以为所管理的Scheduler实例注册相应的Trigger、Calender等系列“装备”。当然，大部分时候，或许只是注册必要的Trigger而已，如下所示：

```
<bean id="scheduler"➥
class="org.springframework.scheduling.quartz.SchedulerFactoryBean">
    <property name="triggers">
        <list>
            <ref bean="newsTrigger"/>
            <ref bean="simpleTrigger"/>
        </list>
    </property>
</bean>
```

SchedulerFactoryBean所管理的Scheduler将随ApplicationContext的实例化自动启动，随ApplicationContext的关闭自动关闭，所以，在将调度过程中所需要的所有对象实例添加到IoC容器之后，只要实例化SchedulerFactoryBean所在的ApplicationContext，Scheduler即开始执行正式的任务调度，如下代码所示：

```
public static void main(String[] args) {
    ApplicationContext ctx = new ➥
    ClassPathXmlApplicationContext(".../container-configlocation.xml");
    ((AbstractApplicationContext)ctx).registerShutdownHook();
}
```

大部分情况下，我们都会在IoC容器启动之前将所有任务调度相关的对象配备完毕，所以，容器启动之后，并不需要获取SchedulerFactoryBean所管理的Scheduler实例。不过，如果在运行期间依然需要动态注册某些调度任务的话，可以从ApplicationContext中获取SchedulerFactoryBean

对应的对象实例，并强制转型为Scheduler类型，如下所示：

```
ApplicationContext ctx = new➥
ClassPathXmlApplicationContext("../container-config-location.xml");
((AbstractApplicationContext)ctx).registerShutdownHook();

Scheduler scheduler = (Scheduler)ctx.getBean("scheduler");
scheduler.addJob(newJobDetail, true);
// 如果必要，使用scheduler做进一步操作……
```

至于为什么将SchedulerFactoryBean强制转型为Scheduler，我想就不必解释了。如果你不清楚，那么回头看一下FactoryBean的特性。

最后，我们给出的是FXNewsProvider对应的任务调度相关配置片段，如代码清单31-5所示。

代码清单31-5 FXNewsProvider相关任务调度类配置代码示例

```
<!--// 其他可能依赖的bean定义-->
<beanid="fxNewsProvider"➥
class="org.darrenstudio.books.unveilspring.news.FXNewsProvider"➥
p:newsListener-ref="newsListener" p:newPersistener-ref="newsPersister"/>

<bean id="jobDetail" class="org.springframework.scheduling.quartz.➥
MethodInvokingJobDetailFactoryBean">
    <property name="targetObject" ref="fxNewsProvider"/>
    <property name="targetMethod" value="getAndPersistNews"/>
</bean>

<bean id="newsTrigger"➥
class="org.springframework.scheduling.quartz.CronTriggerBean">
    <property name="jobDetail" ref="jobDetail"/>
    <property name="cronExpression" value="0 0/1 * * * ?"/>
</bean>

<bean id="scheduler"➥
class="org.springframework.scheduling.quartz.SchedulerFactoryBean">
    <property name="triggers">
        <list>
            <ref bean="newsTrigger"/>
        </list>
    </property>
</bean>
```

这或许可以帮助你回顾一下从"原生态的Quartz"走到"Spring中的Quartz"这整个旅程。实际上，如果要添加新的调度任务的话，基本上也就是在重复以上这一配置过程，只不过每一调度任务特定的细节还是需要我们时刻关注的。

31.2 Spring对JDK Timer的集成

31.2.1 JDK Timer小记

31

Timer是随JDK 1.3的发布而引入的任务调度"标准装备"，但无论是在设计与实现上，还是在整体的功能支持上，Timer都无法与Quartz相媲美。Timer没有明确的Trigger的概念，一般意义上来讲，Timer自身肩负着Scheduler和Trigger的双重使命，而且，不像Quartz可以通过cron表达式指定复杂的调度规则。Timer基本上只提供SimpleTrigger类似的调度行为支持，所以，大部分时间，我们会优先考虑使用Quartz作为任务调度框架。不过，世事无绝对，如果当前的任务调度需求十分简单，而

且，在使用第三方类库有限制的情况下，Timer也不失为一种经济实惠的选择。

使用Timer进行任务调度要比使用Quartz简单许多，同样的FXNewsProvider功能调度，采用Timer实现只涉及少许的迁移。因为所有经由Timer进行调度的任务都需要继承TimerTask，所以，我们首先需要提供FXNewsProvider对应的TimerTask实现，如代码清单31-6所示。

代码清单31-6　FXNewsProvider对应的TimerTask实现类代码示例

```
public class FXNewsTimerTask extends TimerTask {
    private FXNewsProvider fxNewsProvider;
    @Override
    public void run() {
        getFxNewsProvider().getAndPersistNews();
    }
    public FXNewsProvider getFxNewsProvider() {
        return fxNewsProvider;
    }
    public void setFxNewsProvider(FXNewsProvider fxNewsProvider) {
        this.fxNewsProvider = fxNewsProvider;
    }
}
```

现在，我们应该可以感受到将业务逻辑尽可能剥离到独立业务对象的好处。如果当初直接将要调度的业务逻辑写入Job实现类，那么现在或许又得重蹈覆辙，在TimerTask实现类中重新复制一遍了。

不管怎样，在有了FXNewsTimerTask之后，我们就可以通过相应的Timer实例对其进行调度，如下代码所示：

```
FXNewsTimerTask task = new FXNewsTimerTask();

Timer timer = new Timer("schedulerName[optional]");
timer.schedule(task, 0, 60000);
```

喔！看起来这要比使用Quartz简单多了。不过，就跟使用编程方式演示Quartz的使用一样，这里的代码也只是原则上按照规则办事，而实际上，在FXNewsTimerTask没有获得相应的依赖注入之前，整段代码是无法正常运行的，所以，如你所愿，我们马上进入Spring对Timer的集成内容。

31.2.2　Spring 集成后的 JDK Timer

1. 逃离TimerTask的魔咒

原则上讲，任何将由Timer进行调度的任务都需要继承自TimerTask，说把相应的业务对象压得喘不过气来显然有点儿夸张了，但在这个处处讲求“非强制，少侵入”的时代，这一强制要求显然也不是很受人欢迎。不过，好在我们可以将这项硬性规定的实现转给其他人代为“打理”，这样，原本就是自由身的业务对象可以继续无拘无束，而这个代替业务对象行使TimerTask职责的人，则可以是Spring提供的MethodInvokingTimerTaskFactoryBean。

实际上，将为Quartz提供的MethodInvokingJobDetailFactoryBean与MethodInvoking-TimerTaskFactoryBean做对比，也就不难理解MethodInvokingTimerTaskFactoryBean的存在目的了。在我们为某一业务对象披上MethodInvokingTimerTaskFactoryBean这一外衣之后，它就真得看起来是一个TimerTask了，例如：

```
<beanid="fxNewsProvider"➥
class="org.darrenstudio.books.unveilspring.news.FXNewsProvider"➥
p:newsListener-ref="newsListener" p:newPersistener-ref="newsPersister"/>
<bean id="task" class="org.springframework.scheduling.timer. ➥
```

```
MethodInvokingTimerTaskFactoryBean">
    <property name="targetObject" ref="fxNewsProvider"/>
    <property name="targetMethod" value="getAndPersistNews"/>
</bean>
```

现在的MethodInvokingTimerTaskFactoryBean将代替FXNewsProvider作为一个TimerTask，只是在真正需要执行调度逻辑的时候，才会根据我们指定的信息，通过反射来调用指定实例上的指定方法。显然，MethodInvokingTimerTaskFactoryBean加上FXNewsProvider，给出了完成FXNewsTimerTask同一功能的另一种选择，而至于最终选择哪种，那看来得因人而异了。

2. TimerTask的模块化封装——ScheduledTimerTask

传统的TimerTask对应的Trigger信息是在将TimerTask提交Timer调度的时候指定的，如下所示：

```
timer.schedule(task, 0, 60000);
```

每一TimerTask与其对应的Trigger信息之间没有一种紧密的关联关系。Spring提供了ScheduledTimerTask，对TimerTask和与之相关的Trigger信息进行模块化封装，使得我们可以一种统一的方式来管理TimerTask及其对应的Trigger信息。

因为我们的FXNewsProvider通常是每隔一分钟调度一次，而其对应的TimerTask我们也通过MethodInvokingTimerTaskFactoryBean定义好了，现在，我们就可以通过一个ScheduledTimerTask，将其对应的TimerTask和相关的Trigger信息封装到一个可以统一管理的实体中，如下所示：

```
<bean id="scheduledTask" ➥
class="org.springframework.scheduling.timer.ScheduledTimerTask">
    <property name="timerTask" ref="task"/>
    <property name="period" value="3000"/>
</bean>
```

在有了封装了所有TimerTask和相关Trigger信息的ScheduledTimerTask实例之后，Timer就可以根据ScheduledTimerTask所提供的信息，对相应的TimerTask进行调度了。

3. Timer的新家——TimerFactoryBean

Spring通过TimerFactoryBean对JDK的Timer进行封装。在我们将“以ScheduledTimerTask形式封装好的调度的任务”提交给TimerFactoryBean之后，TimerFactoryBean内部将提取每一ScheduledTimerTask所包含的TimerTask以及相关的Trigger信息，从而决定调用Timer的哪个调度方法，对当前ScheduledTimerTask所包含的TimerTask进行调度。整个过程如代码清单31-7所示。

代码清单31-7　TimerFactoryBean实现代码摘录

```
ScheduledTimerTask[] stt = ...;
for(ScheduledTimerTask task:stt)
{
    if (task.isOneTimeTask()) {
        timer.schedule(task.getTimerTask(), task.getDelay());
    }
    else
    {
        if (task.isFixedRate()) {
            timer.scheduleAtFixedRate(task.getTimerTask(), task.getDelay(), task. ➥
            getPeriod());
        }
        else {
```

31

```
            timer.schedule(task.getTimerTask(), task.getDelay(), task.getPeriod());
        }
    }
}
```

所以，现在只要将我们的NewsProvider对应的ScheduledTimerTask实例提交给TimerFactory-Bean，使用Timer对FXNewsProvider功能的调度就算大功告成，如下所示：

```
<bean id="scheduler"➥
class="org.springframework.scheduling.timer.TimerFactoryBean">
    <property name="scheduledTimerTasks">
        <list>
            <ref bean="scheduledTask"/>
        </list>
    </property>
</bean>
```

我们回顾一下Spring为集成Timer而提供的各类装备，脑海中或许应该浮现如图31-4所示的一副图景。

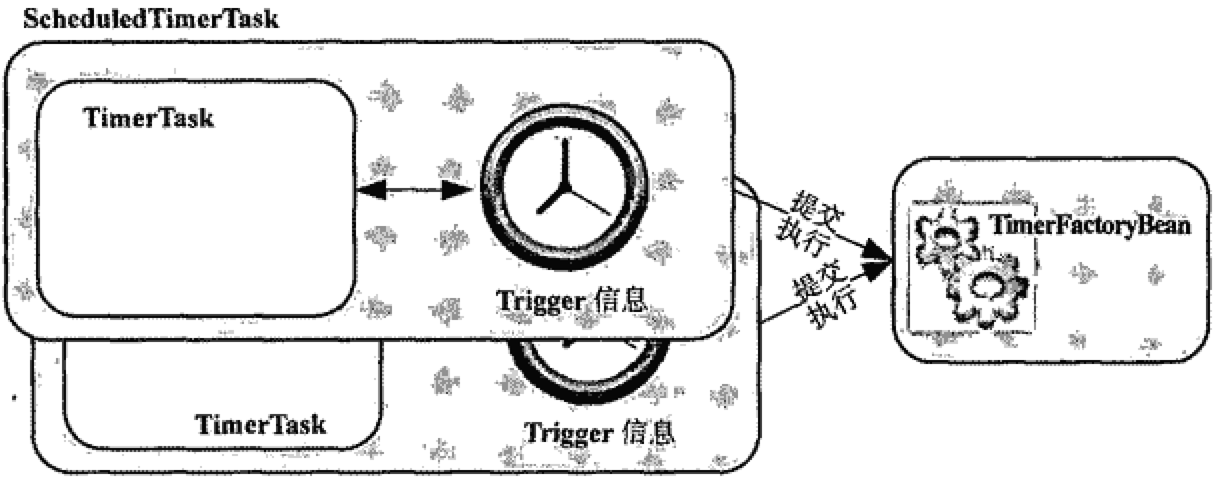

图31-4　Spring中的Timer相关概念实体关系图

当然，最主要的是，在我们使用了Quartz和Timer两种任务调度方案对同一功能的调度给予实现之后，通过对比这两种方案，你又有何想法呢？

31.3　Executor 的孪生兄弟 TaskExecutor

Java 5为我们带来了一套崭新的任务执行框架（task execution framework），该框架以Executor为首：

```
public interface Executor {
    void execute(Runnable task);
}
```

虽然该顶层抽象接口Executor看起来是那么简单，但所带来的好处却并不简单。Executor这一抽象的意义在于，可以将任务的提交（task commission）和任务的执行（task execution）策略分隔开来，解除二者之间的耦合性，以Runnable类型界定的任务提交之后，最终会以什么样的策略执行（什么时间执行，交给谁执行，等等），完全由不同的Executor实现类负责，提交任务的客户端完全可以忽略后者的实现细节，所以，虽然大部分时间我们会使用像ThreadPoolExecutor之类提供线程池（Thread Pool）且以异步方式执行任务的Executor实现，但Executor自身的语义却并不只限于线程池。

Executor任务执行框架提供了许多功能强劲的Executor实现类，但是，这些都是基于Java 5的。如果我们要在之前版本的Java平台上使用的话，显然就得做一些兼容性的工作，而Spring 2.0之后提出的TaskExecutor，实际上就是为了这个目的而生的。现在，我们可以在Java 1.4或者Java 5上使用TaskExecutor完成Executor同样的使命，因为二者的定义实际上没有什么实质上的差别，TaskExecutor如下定义：

```
public interface TaskExecutor {
    void execute(Runnable task);
}
```

当然，最主要的是，Spring为TaskExecutor提供了足够的实现类选择，而且，即使TaskExecutor没有相应的实现类与某个Executor提供的功能相对应，Spring也提供了“飞架TaskExecutor和Executor的桥”，这些应该让TaskExecutor的使用没有任何后顾之忧。

31.3.1 可用的 TaskExecutor

Spring框架内部提供了一定的TaskExecutor实现类，这些实现类可以完成几乎所有使用场景的覆盖，所以，大多数情况下，我们没有必要自己去实现某个TaskExecutor。即使我们真的需要去实现新的TaskExecutor，也应该首先来了解一下Spring都为我们提供了哪些现有的TaskExecutor实现吧？

1. SyncTaskExecutor

提交给SyncTaskExecutor的任务将直接在当前调用线程中执行，所以该类的定义也是出奇地简单，如下所示：

```
public class SyncTaskExecutor implements TaskExecutor, Serializable {
    public void execute(Runnable task) {
        Assert.notNull(task, "Runnable must not be null");
        task.run();
    }
}
```

更多时候，我们只会在测试的时候想到它。

2. SimpleAsyncTaskExecutor

SimpleAsyncTaskExecutor提供的任务执行策略十分简单和直观，它提供最基础的异步执行能力，而实现的方式则是为每个任务都创建新的线程，所以，原型代码看起来是如下的形式：

```
public class SimpleAsyncTaskExecutor implements TaskExecutor {
    public void execute(Runnable task) {
        new Thread(task).start();
    }
}
```

不过，SimpleAsyncTaskExecutor的实现实际上比我们的原型代码要复杂一些。起码，SimpleAsyncTaskExecutor提供了相应的属性，以控制创建的线程数目上限，如下所示：

```
<bean id="simpleAsycTaskExecutor"➥
class="org.springframework.core.task.SimpleAsyncTaskExecutor">
    <property name="concurrencyLimit" value="100"/>
</bean>
```

如果不做明确设置，SimpleAsyncTaskExecutor对于可以创建的线程数目没有任何限制。

如果线程的创建和销毁不需要占用相应资源，我想，到SimpleAsyncTaskExecutor我们就可以打

住了。不过，事实却是，为了避免线程创建和销毁带来的开销，我们需要某个使用线程池提供异步任务执行能力的TaskExecutor，所以，接着往下看吧！

3. ThreadPoolTaskExecutor

既然SimpleAsyncTaskExecutor每次都要创建新的线程的做法不可取，ThreadPoolTaskExecutor改用线程池来管理并重用处理任务异步执行的工作线程。实际上，ThreadPoolTaskExecutor只是对标准的java.util.concurrent.ThreadPoolExecutor进行了封装。只不过可以通过类似JavaBean的风格进行配置。当然，封装后的java.util.concurrent. ThreadPoolExecutor，最终还是以TaskExecutor类型公开给客户端代码使用的。

Spring提供的ThreadPoolTaskExecutor一共有两个，一个是org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor，它对Java 5的java.util.concurrent.ThreadPoolExecutor进行封装；另一个是org.springframework.scheduling.backportconcurrent.ThreadPoolTaskExecutor，它对JSR-166 backport的ThreadPoolTaskExecutor进行封装。如果我们的应用程序需要在Java 1.4平台运行，那么后者才是我们应该选用的TaskExecutor类型。

ThreadPoolTaskExecutor公开了一些配置属性允许我们对使用的ThreadPoolTaskExecutor进行定制，比如：

```
<bean id="threadPoolTaskExecutor" class="org.springframework.scheduling.concurrent. ➥
ThreadPoolTaskExecutor">
    <property name="corePoolSize" value="10"/>
    <property name="maxPoolSize" value="20" />
    <property name="queueCapacity" value="20" />
</bean>
```

更多定制属性及相关信息可以在使用的时候参照相应类的Javadoc。通常情况下，ThreadPoolTaskExecutor将是我们的首要选择。

4. ConcurrentTaskExecutor

ConcurrentTaskExecutor为Java 5的Executor和Spring的TaskExecutor搭建了一道桥梁，使得我们可以将Executor框架下的某些实现类以TaskExecutor的形式公开出来。如果我们感觉ThreadPoolTaskExecutor封装的java.util.concurrent.ThreadPoolExecutor不足以满足当前场景需要，那么可以构建需要的Executor实例，比如通过Executors.newXXXThreadPool()，然后以ConcurrentTaskExecutor对其进行封装，封装后获得的ConcurrentTaskExecutor即获得相应Executor的能力，但它现在是以TaskExecutor的样子示人[①]，如下所示：

```
Executor executor = Executors.newScheduledThreadPool(10);
TaskExecutor taskExecutor = new ConcurrentTaskExecutor(executor);
```

如果我们使用Spring的IoC容器来配置使用ConcurrentTaskExecutor，那么可以直接配置它所封装的Executor实例，或者通过FactoryBean机制对它所使用的Executor实例做进一步定制。

① 在如下代码示例中，Executors.newScheduledThreadPool()返回的实际是一个ScheduledThreadPoolExecutor实例。因为JDK Timer在任务调度的时候存在一些问题，比如因某一轮任务执行延迟导致后继执行没能够即时触发，或者如果调度中某个任务执行期间抛出运行时异常，可能导致后继任务无法被调度，所以，在Java 5之后，既有Timer身手，又没Timer缺陷的ScheduledThreadPoolExecutor就成为了JDK Timer的继任者。如果应用程序构建在Java 5或者更高版本的平台上，那么，即使Spring提供了Timer的任务调度支持，也不要考虑使用Timer了。

> **注意** 与Spring的ThreadPoolTaskExecutor一样，ConcurrentTaskExecutor也存在两个版本，一个面向Java 5，一个面向JSR-166 backport，使用的时候可不要混淆哦！

5. **TimerTaskExecutor、SimpleThreadPoolTaskExecutor和WorkManagerTaskExecutor**

之所以将这三个实现类放到一起，是因为它们实在是太相似了，因为它们都是使用特定的调度程序（Job Scheduler）来执行提交给它们的任务。

TimerTaskExecutor内部持有一个Timer的引用，在任务提交给TimerTaskExecutor执行之后，TimerTaskExecutor将对Runnable类型提交的任务以TimerTask形式进行包装，然后就转交给它所持有的那个Timer来调度执行了。所以，简单点儿来看，TimerTaskExecutor的原型代码可以如下构建：

```
public class PrototypeTimerTaskExecutor implements TaskExecutor{
    private Timer timer = new Timer();
    public void execute(final Runnable task) {
        timer.schedule(new TimerTask(){
            @Override
            public void run() {
                task.run();
            }}, new Date());
    }
}
```

实际的TimerTaskExecutor实现做了更多的关注点分离，并提供了更详细的设定选项，以Runnable提交的task实际上是以org.springframework.scheduling.timer.DelegatingTimerTask的形式单独封装的，而不是匿名内部类，而像执行延时（delay）之类属性也是可配置的。

SimpleThreadPoolTaskExecutor扩展自Quartz的SimpleThreadPool，它使用SimpleThreadPool的runInThread(task)方法来执行提交的任务。我想，如果不在意绑定到Quartz，或者想将SimpleThreadPoolTaskExecutor提供给Quartz以替换默认的SimpleThreadPool使用，通常应该不会考虑使用SimpleThreadPoolTaskExecutor吧！

CommonJ是BEA和IBM联手搞得一套应用服务器端的任务调度实现规范。WorkManagerTaskExecutor就是使用CommonJ的WorkManager构建的。至于什么场合下使用它，我想你应该可以猜得到。

31.3.2 TaskExecutor 使用实例

假设我们的FXNewsProvider在通过FTP接收到新闻文件之后，想对接收到的文件进行并行处理，以提高处理速度，那么，我们可以考虑将单一新闻文件的处理逻辑进行Runnable形式的任务封装，然后提交给相应的TaskExecutor进行处理。为了达到这一目的，我们首先要对新闻文件解析和持久化的逻辑进行封装，封装后的代码如代码清单31-8所示。

代码清单31-8 封装新闻处理的Runnable任务实现类代码示例

```
public class NewsProcessingTask implements Runnable {
    private IFXNewsListener listener;
    private IFXNewsPersister persister;
    private String newsId;

    public NewsProcessingTask(IFXNewsListener listener,IFXNewsPersister persister,String➥
    newsId)
    {
        this.listener  = listener;
```

31

```
        this.persister = persister;
        this.newsId    = newsId;
    }
    public void run()
    {
        FXNewsBean newsBean = getListener().getNewsByPK(newsId);
        getPersister().persistNews(newsBean);
        getListener().postProcessIfNecessary(newsId);
    }
}
```

然后，为FXNewsProvider提供一个TaskExecutor以执行相继提交的NewsProcessingTask。为了区别于之前的FXNewsProvider，我们称现在的实现为RefactoredFXNewsProvider①，其定义如代码清单31-9所示。

代码清单31-9　重构后使用了TaskExecutor的FXNewsProvider实现类代码示例

```
public class RefactoredFXNewsProvider
{
    private IFXNewsListener newsListener;
    private IFXNewsPersister newPersistener;

    private TaskExecutor taskExecutor;

    public void process()
    {
        String[] newsIds = newsListener.getAvailableNewsIds();
        if(ArrayUtils.isEmpty(newsIds))
        {
            return;
        }

        for(String newsId : newsIds)
        {
            NewsProcessingTask task = new NewsProcessingTask(getNewsListener(),➥
            getNewPersistener(),newsId);
            getTaskExecutor().execute(task);
        }
    }
    // getter和setter方法定义
}
```

现在，我们只要为RefactoredFXNewsProvider提供必需的依赖就可使之开始工作。假设我们为其提供ThreadPoolTaskExecutor作为使用的TaskExecutor，那么对应的配置内容如代码清单31-10所示。

代码清单31-10　RefactoredFXNewsProvider相关类配置代码示例

```
// 其他可能依赖的bean定义……

<bean id="taskExecutor"
class="org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor">
```

① 从FXNewsProvider到RefactoredFXNewsProvider重构，我们可以看到当初没有使用面向接口编程所带给我们的些许尴尬：原来依赖于FXNewsProvider的客户端代码现在或许同样要受此重构之牵连，而如果当初提供抽象接口，那么现在只是提供新的实现类，然后直接替换掉旧的实现类而已。不过，要是直接在FXNewsProvider上动手术，也可以避免对客户端代码的牵连吧！

```
        <property name="corePoolSize" value="10"/>
        <property name="maxPoolSize" value="20" />
        <property name="queueCapacity" value="20" />
    </bean>

    <bean id="rFxNewsProvider" class="..RefactoredFXNewsProvider">
        <property name="newsListener" ref="newsListener"/>
        <property name="newPersistener" ref="newPersistener"/>
        <property name="taskExecutor" ref="taskExecutor"/>
    </bean>
```

实际上，只要声明依赖的是TaskExecutor接口，最终使用哪一种TaskExecutor实现（实际上也正是选择任务的执行策略）完全可以在部署的时候决定，无非只是配置文件的些许改动而已。

31.4 小结

本章我们对Spring框架提供集成支持的几种任务调度程序进行了“身世”分析，这包括Quartz、JDK Timer以及Spring提供的TaskExecutor。在阅读完本章内容之后，你应该可以根据自身应用的情况，做出最终的选择。

第32章

Spring框架对J2EE服务的集成之扩展篇

本章内容

- MailMonitor的延伸
- Spring 3.0展望

32.1 MailMonitor 的延伸

你或许会纳闷？什么是MailMonitor？从它能够延伸出什么？下面让我们一起来揭开这个谜底。

MailMonitor是我曾经参与的信贷项目（CREDIT）中的一个批处理程序。该批处理程序将检查CREDIT系统向顾客发送的各类邮件是否发送成功，如果发现邮件标题中含有发送失败的信息关键字，则对这些错误邮件进行接收并分析，将分析后的信息存入各种数据表，然后再将处理后的邮件删除。可以说整个MailMonitor批处理的需求和实现都很简单。之所以它拿开刀，是因为它的应用需求并不难理解，而且，“麻雀虽小，五脏俱全”。对MailMonitor的剖析和重构，可以让我们从整体上回顾一下Spring对各种Java EE服务的集成支持。

俗话讲，“家丑不可外扬”。不过，如果能够从过去吸取教训并进步，也不算坏事，所以，我就把04年的代码晾晒于下（见代码清单32-1），然后让我们一起对其进行一番批斗，如何？[①]

代码清单32-1　最初的MailMonitor调度任务实现类代码示例

```
public class MailMonitorJob extends QuartzJobBean {
   private static final Logger logger = Logger.getLogger(MailMonitorJob.class);

   private ServletAuthUtils authUtils;
   private List keyWord;

   private IMonitorRequestProcessor requestProcessor;

   protected void executeInternal(JobExecutionContext context) ➥
   throws JobExecutionException {
        Store store = null;
        Folder f = null;
         try
         {
             Properties prop = System.getProperties();
```

① 原先的代码整体上看起来过长，所以笔者在不丧失主要关注点的情况下，对其做了一定的剪裁。

```
            Session session = Session.getDefaultInstance(prop,null);

            ServletAuthResult result = authUtils.auth();
            String host = result.getSmtpServer();
            String user = result.getSmtpUser();
            String password = result.getSmtpPassword();

            URLName url = new URLName("pop3",host,-1,"INBOX",user,password);

            store = session.getStore(url);
            store.connect(host,user,password);

            f = store.getFolder("INBOX");
            if(f == null || !f.exists())
            {
                return;
            }
            f.open(Folder.READ_WRITE);
            if(!f.isOpen())
            {
              return;
            }

            Message[] messages = f.getMessages();
            if(ArrayUtils.isEmpty(messages))
            {
              if(logger.isInfoEnabled())
                   logger.info("No Message Exists in the mail-list!");
              return;
            }
            processFailureMessages(messages);

        }
        catch(Exception e)
        {
            throw new JobExecutionException(e);
        }
        finally
          {
          closeFolder(f);
          closeStore(store);
          }
    }
    // 其他helper方法定义
}
```

在审视这段代码的时候，我们可以发现以下这些问题或者需要改进的地方，甚至更多，如下所述。

- 我们让批处理任务实现类直接继承了Spring的QuartzJobBean。在介绍Spring对Quartz的集成的时候，我们提到，虽然QuartzJobBean可以让我们的批处理任务实现类获得配置的好处，但其极强的侵入性使得它的存在已经不合时宜。显然，让我们的MailMonitorJob脱离QuartzJobBean是我们重构所需要做的第一个工作。
- 邮件接收和邮件处理相关功能关注点没有进行合理的分离，JavaMail API的使用代码与邮件处理相关代码相互纠缠，这使得整段代码的重用性和可扩展性极差。为了改善这种状况，我们可能需要做如下工作。

32

- 剥离使用JavaMail API实现的邮件接收功能到独立的Helper类，使得该Helper类可以提供可重用的邮件接收基础设施代码。Spring提供了MailSender和JavaMailSender等基础设施用于邮件的发送，可并没有提供对邮件接收功能的集成设施，我们的Helper就是要完成这一功能。而至于如何来实现这个用于邮件接收的Helper类，参照之前Spring给予我们的经验或者JavaMailSender的实现，我想你心中早就有数了。虽然邮件信息接收后的解析和处理不好统一，但是API相关的资源管理等都可以模板化。至于最终实现，我就不再啰嗦了。
- 最初的MailMonitor实现，是将发送失败的邮件信息的处理逻辑，整个放在了MailMonitorJob内实现，而且处理邮件信息的方式也只是简单的循环依次处理，如下所示：

```
Message[] messages = ...; // 我们接收到的message
for(Message message:messages)
{
  // 解析message
  // 持久化存储解析后的message信息
  // 从邮件服务器删除该message
}
```

对这种情况的重构，我们可以引入TaskExecutor来解耦任务的提交和执行，这不但可以将邮件处理逻辑以任务（Task）的形式进行合适粒度的封装，而且还可以引入灵活的任务处理策略，何乐而不为呢？

如果我们能够从这几个方面对最初的MailMonitor进行重构，我们基本就完成了从Spring的MailSender抽象层到任务调度和TaskExecutor的基本回顾，牵强一点儿，再让JavaMail的Session从JNDI获取，那么也就剩下JMS没有顾及到了。但是我想，如果我们能够从MailMonitor这只“麻雀”的剖析中引申出去的话，是不是可以发现更加广阔的天地呢？实际上，这也正是我所希望的！

32.2　Spring 3.0 展望

Java 5中，JSR 166（Concurrency Utilities）的引入，对于Java平台来说可谓意义深重。Spring 3.0中将继续倾注更多热情在对并发的支持上面，这包括：

- 更新TaskExecutor抽象接口，以便更好地与现有的ava.util.concurrent进行集成；
- 为Callable和Future提供一等公民待遇，为ExecutorService和ThreadFactory提供相应的集成支持，以便它们能够更好的融入Spring容器的怀抱；
- 提供异步方法调用支持，这可以通过Spring 3.0新添加的@Async或者EJB 3.1中的@Asynchronous进行触发。

除此之外，Spring 3.0中还可能为任务调度（scheduling）提供独立的命名空间（namespace），这将大大简化相应的配置。最后要提到的就是，Spring 3.0中会提供使用cron表达式的Timer。至于长什么样子，我们只能“走着瞧”了。

32.3　小结

本章主要想就着对MailMonitor的剖析和重构，帮助你进一步巩固本部分Spring对各种Java EE服务的集成相关内容。

第33章 Spring远程方案

本章内容

- 从“对面交谈”到“千里传声”
- Spring Remoting架构分析
- Spring Remoting提供的远程服务支持
- 扩展Spring Remoting
- Spring Remoting之扩展篇

33.1 从“对面交谈”到“千里传声”

话说某日，八神[①]突发要跟草稚对决之念头，便直接走上街头去堵草稚。在二人面对面站定之后，八神便直接向草稚发话，要求决斗云云，草稚当然可以接受其挑战，也可以拒绝之。整个场景用图说话可能更具说服力一些，如图33-1所示。

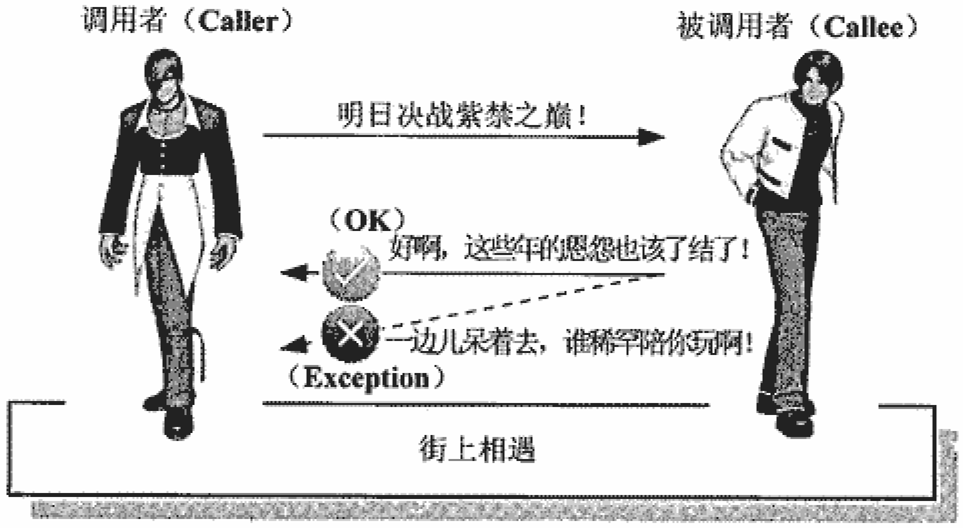

图33-1 对决之“本地调用”版

不要误会，我不是在讨论《拳皇》，实际上是要讨论对象间的调用。确切地说，是本地调用。你难道不觉得图33-1所展示的场景跟对象间的本地调用很相似吗？因为八神对象需要请求草稚对象做什么事情的时候可以直接找到对方（在本地调用中就相当于能够直接获得对方的引用），所以直接发送“对决”调用请求即可。至于调用的结果，那完全要看被调用的草稚对象如何处理调用请求了。如果

① 八神和草稚是《拳皇》里的人物，我想大家应该会或多或少接触过或者有所耳闻，所以才以他们之间的关系做比喻（稍加夸张）。如果读者之前确实没有类似的经历，不妨上网搜索一下他们之间的故事吧！

他答应八神对象的“对决”调用请求，那相当于调用正常返回；相反，如果回绝，那跟我们的业务对象通过抛出业务相关异常来表示调用失败又是多么地相似。

大部分时间，我们都是在开发这种由本地调用组成的软件系统，甚至于在继续余下内容之前，Spring框架为我们所提供的各种帮助也大多是局限于此。不过，并不是什么事情都会按照我们的意愿行事，当八神再次想与草稚对决的时候，情况发生变化了……

八神当然希望能够面对面地跟草稚挑明，但不巧的是，草稚现在不在当前进程中，即不在本地，他现在身在异乡（如图33-2所示）。这种情况下，八神该怎么跟草稚进行交流呢？不嫌慢并且草稚那边有固定地址，那写信吧！如果等不及，并且草稚现在没有固定地址，那有电话也行啊！什么？电话也没有，那能上网不？MSN或者Skype都行，甚至于不管什么手段，只要能跟草稚对象联系上，把对决的请求发送给他然后获得相应答复就行啊！

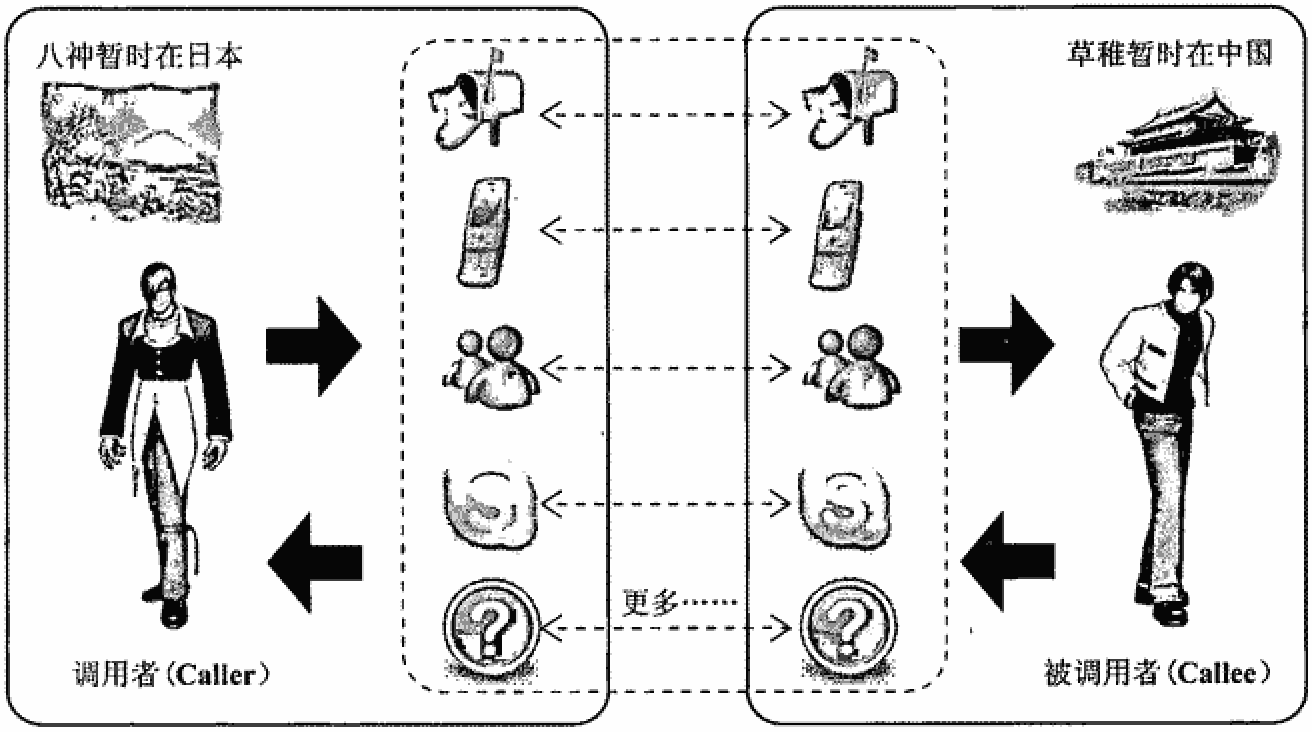

图33-2　对决之“远程调用”版

你看，就因为两个对象相隔太远，只要它们需要交流，我们就不得不想尽各种办法把它们联系到一起。现实世界中，我们可以通过电话，IM（Instant Messaging）软件等。当把草稚和八神化作两个位于不同进程，或者不同主机上运行的对象的时候，我们就得使用其他方式，比如RMI、EJB、Web服务等手段，现在，我们进入了分布式应用的地盘。

像草稚和八神这种，因为地域界限（映射到远程访问就是进程间或者主机间的界限）而不得不采用各种远程方式进行相互访问的情况，我们可以将其看作是“被逼无奈”之举。如果位于中国的某个应用需要访问位于美国某台主机上的某种服务，那么它就不得不采用某种远程方式来完成服务的访问；但对于某些应用来说，引入某些远程机制从而实现分布式架构，却是它们本身“自愿”的，因为它们可以从分布式架构中获取它们需要的东西，比如，通过负载均衡以扩展系统性能，通过多台并行主机让系统获得“灾难恢复”的能力，通过统一部署简化系统维护，等等。当然，我们可以列举更多，但是，任何东西的获取都是有“机会成本”存在的。分布式架构带给我们的好处也同样需要付出相应的代价，最为突出的问题就是整个系统的复杂度突增。相对于本地调用来说，我们现在要饱受网络延迟之苦，要考虑并处理多个请求对同一对象（或者组件）的并发访问问题。如果系统出现故障，我们大多无法处理，因为既可能是网络问题，也可能是被访问端出现故障。总之，现在我们要考虑太多太

多，这或许也正是为什么Martin Fowler提出的分布式对象第一原则是"不要分布你的对象"的原因吧！

那是不是因为分布式架构的复杂度，我们就真得不去分布我们的对象了呢？这应该由应用场景来决定。既然分布式架构利弊都有，我们就需要权衡。当我们的应用能够从分布式场景中获得更多的"利"的时候，采用分布式架构是合理的，而只能获得少量的"利"，甚至"微利"的时候，采用分布式架构反而是不合适的。

对Java开发人员来说，一提到分布式架构，第一个跃入脑海的多半就是EJB（我指的是EJB 1.x～2.x）。应该说，在处理分布式需求方面，EJB表现还是不错的，部分功能方面的设计失败，并不能掩盖EJB在分布式处理方面的不错表现，但是，为什么现在EJB受人诟病那么多呢？EJB中部分功能设计，比如实体Bean（Entity Bean），确实为败笔，但更多时候，EJB的诟病却是由其提供分布式服务的本质而招致的。分布式架构本身就会为整个应用引入过多的复杂度，所以，对分布式功能提供支持的EJB自然也就简单不到哪儿去。在我们倡导Without EJB的开发模式的时候，更多是因为我们所处的应用场景并不需要引入分布式的概念，Spring提倡的轻量级开发方案不是彻底地否定EJB，而是否定对EJB的滥用，或者更广一些，对各种技术方案在不恰当场景中的滥用。

如果按照"帕累托法则"（Pareto Principle，也叫80/20法则）进行划分的话，我们日常所开发的应用大都属于不需要分布式架构的那80%之列，那么，正常情况下，我们就不应该引入分布式的概念，但是对EJB的滥用却使得这80%的净土也让分布式的复杂度所污染。这就好像你本来不需要的东西，别人一定要硬塞给你一样，到时候沉的拿不动还是小事，还要跟你要银子的话，那岂不更惨？难道当年那些不需要EJB提供分布式支持的应用，被硬是扣上EJB这顶沉重的帽子，与此不是很相似吗？鉴于此，Spring才提出Without EJB的理念，Spring的开发理念希望我们轻装上阵，在系统设计和开发之初，只需要关注核心业务之类关注点，对于像分布式之类的需求暂且扔到一边，待需要用的时候再添加也不迟。况且，只要设计良好，为现有系统添加分布式访问能力也并不像我们想象的那么棘手，尤其是在有了Spring Remoting的相关框架类支持之后，完成从本地调用到远程调用的转变变得更是得心应手，所以，还是赶快开始我们的Spring Remoting之旅吧！

33.2 Spring Remoting架构分析

为了帮助我们简化公开和访问远程服务的相关工作，Spring Remoting推出了"Service Exporter和Service Accessor组合"。通过该魅力组合，我们可以统一的风格将本地服务以远程的形式公开出去，也可以统一的风格来访问远程服务。在Spring Remoting为我们准备好了针对不同远程机制的"Service Exporter和Service Accessor组合"之后，我们所要做的，也就剩下简单的配置工作了。

除了核心的"Service Exporter和Service Accessor组合"，Spring Remoting同时也考虑到了远程访问过程中可能发生的各种访问异常，并给出了一套标准远程访问异常体系。在我们详细了解核心的"Service Exporter和Service Accessor组合"之前，不妨先把这套远程访问异常体系简单看一下吧！

33.2.1 Spring Remoting之远程访问异常体系

你或许会疑惑，远程访问异常是哪一类异常啊？对于本地调用来说，我们从来就不会操心这些，因为它在那种场景下并不存在，远程访问异常纯粹是引入分布式架构，需要通过各种远程机制进行服务访问后的产物。当八神对象通过电话这种远程机制调用草稚对象服务方法的时候，有可能电话线路出现故障，也有可能草稚的手机没电或者关机，这不同于本地调用时候草稚对象抛出的业务相关异常。对于远程访问期间出现的这类异常情况，调用端通常无法有效处理。我想电话线路故障，即使八神对着电话一个劲地吼，电话线路也不太可能自动恢复正常工作吧？对于其他远程机制，也是同样的道理。

鉴于我们通常无法有效处理远程访问异常的事实，Spring Remoting框架提出了一套unchecked exception类型的远程访问异常体系，如图33-3所示。

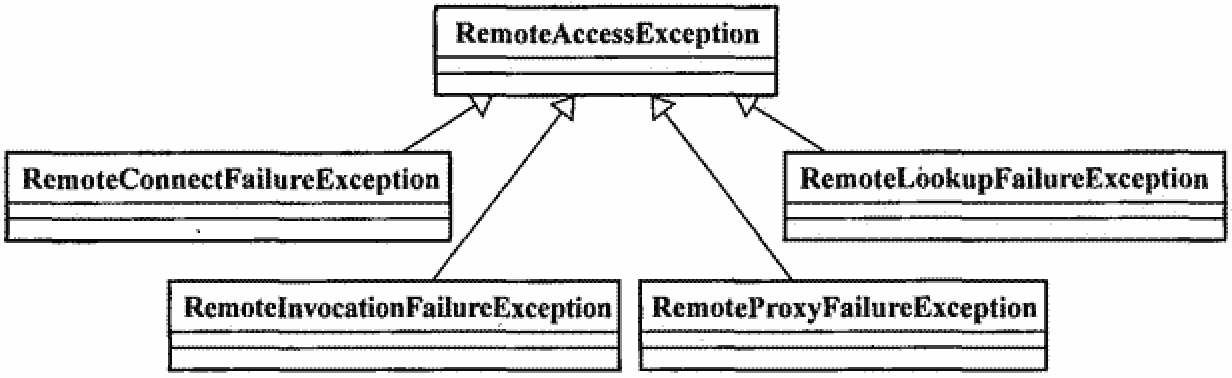

图33-3 Spring Remoting远程访问异常体系关系图

对于这些异常，远程服务调用方既可以任其在调用栈中向上传播，由外层决定如何对其进行处理，也可以自己捕捉以决定如何处置。当然，最主要的，现有的各种远程方案都有自己特定的远程访问异常类型，采用Spring Remoting的这套标准远程访问异常体系之后，各种远程方案特定的异常类型将被转译。这样，远程方案的变动，起码在异常处理这一层次上，不会对调用方造成太大的影响。

33.2.2 统一风格的远程服务公开与访问方式

我们可以在分布式系统中发现多种远程交互方式，比如面向消息中间件（MOM）使用的Message Passing方式，或者数据库管理系统（DBMS）所使用的共享存储区（Shared Repository）方式，但最基本也最广为人知的却应该是RPC（Remote Procedure Call）方式。而Spring Remoting提供的远程服务公开以及访问方式，基本上就是建立在RPC的行为基础上的。

RPC远程交互方式建立在客户端/服务器（Client-Server）模型上。在客户端/服务器模型中，由客户端发起调用请求，请求数据（比如请求的函数名以及参数等）通过相应的协议传输到服务器端（Server）再交给被请求的服务进行处理，最后的处理结果再传送给请求发起的客户端。在处理结果返回之前，客户端通常是阻塞（Blocking）状态。对于一个设计良好的系统来说，不太可能会将远程交互相关的细节全部都附加给业务对象（服务器端的也好，客户端的也好）。如果那样的话，业务对象将陷入远程交互细节之类的泥潭，而无暇顾及真正该其关心的业务逻辑实现。所以，将服务器端以及客户端业务对象通过远程机制进行交互这一关注点，单独剥离到相应的角色来处理，就成为很自然的一件事情。Spring Remoting所做的工作实际上就是如此。

正如我们开始所提到的那样，Spring Remoting最核心的就是它所推出的“Service Exporter和Service Accessor组合”（见图33-4）。

在这一组合中，Service Exporter将主要负责远程服务对象的公开工作，具体点儿说就是，负责根据使用的远程机制（RMI也好，EJB也好，Hessian也好，等等）接收服务请求，然后对请求内容进行解组（un-marshaling）。根据解组后的请求内容调用本地服务对象（对于Client是远程对象的服务对象，对于Exporter来说，就是本地对象了）。调用完成后，将调用结果重新编组（marshaling）然后发送到请求客户端。与此相反，Service Accessor的主要职责当然就是帮助客户端对远程服务对象进行远程访问。在客户端发起调用请求之后，相应的Service Accessor需要对请求内容进行编组，然后根据所使用的远程机制（RMI、Web服务等）对编组后的请求内容进行发送。在接收到Service Exporter发送回的调用结果之后，再对调用结果进行解组，并把解组后的调用结果传给本地的客户端对象使用。

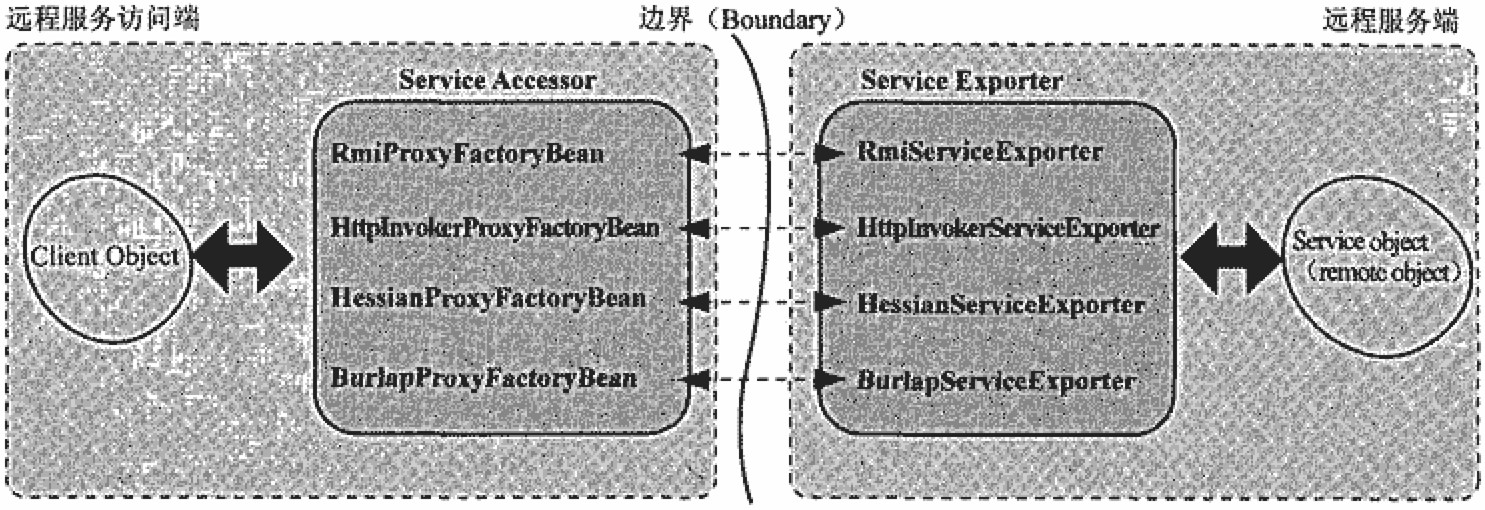

图33-4 Spring Remoting远程交互示意图

在引入“Service Exporter和Service Accessor组合”之后，无论是客户端对象还是远程服务对象都落得个“无远程责任一身轻”，它们只需要关注自己应该关注的业务逻辑即可，对于远程交互之类的事情，就全部扔给具体的Service Exporter和Service Accessor实现类去处理了。这就跟八神和草稚通过电话这种远程机制进行交互一样，至于电话如何将语音信号编组为模拟信号或者数字信号，八神和草稚根本就不需要去关心，他们所要作，只是发话和回话而已。这恰好也正是为什么图33-4与图33-2“对决之“远程调用”版”看起来如此相似，Service Exporter和Service Accessor不就相当于生活场景中的被叫方和主叫方的电话或者MSN等IM客户端吗？

实际上，Spring提供的“Service Exporter和Service Accessor组合”完全就是软件架构中处理远程调用的基本模式的一个缩影。从Spring Remoting对各种远程机制的支持推广开来，我们可以得出一个“几乎”是“放之Remoting世界而皆准”的结论，任何的软件系统，如果它要将本地的服务以远程的形式公开给他人使用，通常都会设置一个类似Service Exporter的角色来负责远程交互的细节。在现有的Remoting模式中，Service Exporter的角色通常称为Invoker；相对地，当本地对象或者组件需要依赖于以远程的形式公开的某一服务的时候，系统中也通常会设置一个类似于Service Accessor的角色负责处理与远程服务的交互工作。在现有的Remoting模式中，Service Accessor的角色通常称为Requestor。不过，Spring Remoting的Service Accessor要比Requestor更进一步。更确切地讲，应该属于Client Proxy，因为它们几乎都是清一色的`ProxyFactoryBean`，可以为本地调用方提供远程服务的代理对象，从而使得本地调用方只需要跟远程服务接口打交道，完全不需要像“过程化编程”时代的RPC那样，需要直接跟Requestor打交道，这使得本地调用或者远程调用的事实对本地调用方来说是完全透明的。

说这么多，唯一的目的就是，当Spring Remoting所提供的对各种远程机制的支持不能满足我们需要的时候，我们完全可以“依葫芦画瓢”，以几乎相同的Remoting模式来实现自己的远程方案。不过，Spring Remoting为现在各种远程机制所提供的支持已经足够丰富了。所以，在考虑自己提供基于某种远程机制的远程方案之前，我们应该首先对Spring Remoting所提供的各种方案给予足够的重视，否则，重新赶工所浪费的人力物力就看起来有些可惜了！

注意 Spring Remoting提供的远程方案通常都是基于无状态的远程服务，这使得整个系统的演化可以更加灵活。如果当前场景“确实”需要有状态的远程交互，那么可以考虑EJB的有状态的Session Bean或者其他方案。但前提一定要明确，只有真正必要的时候才这么做。

33.3 Spring Remoting提供的远程服务支持

在开始了解Spring Remoting提供的各种远程方案支持之前，我们需要有一个可以讨论的基石，你还记得Spring MVC部分中提到的`ITTMRateService`吗？当时我们是通过它提供获取TTM汇率的服务，然后通过web应用的形式将请求结果公开给客户端浏览器，而现在，我们将使用各种远程方案把同一个`ITTMRateService`公开给不同的客户端使用，如图33-5所示。

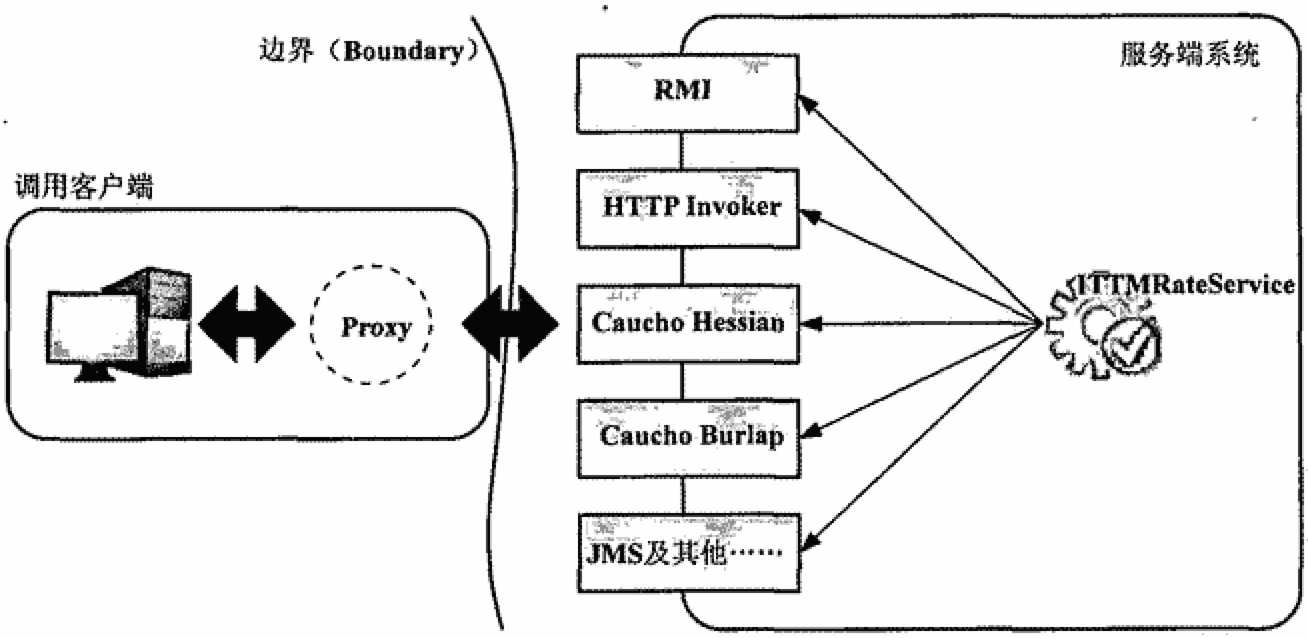

图33-5　`ITTMRateService`远程访问示意图

你将看到，通过Spring Remoting以远程方式公开一个本地服务是多么简单，而客户端应用访问这一远程服务又是多么的省事。

虽然我们当前场景中使用的是单一业务对象的Remoting，但是这完全可以扩展到整个应用的场景中。只要应用程序的业务层设计良好，我们后期完全可以根据不同的客户端类型需求，使用对应的远程机制将业务层提供的服务以远程方式公开出去。整个应用的灵活性和可扩展性可见一斑！从这个角度来讲，我们更应该坚持以业务逻辑为中心的开发准则。无论什么情况下，系统中的业务逻辑都应该以较为独立的形式存在，这些业务逻辑只应该关注自己所关注的东西，依赖自己最需要依赖的对象。而至于谁将使用它们，将以什么样的形式使用它们，不应该是业务逻辑实现所需要关心的事情。我们要对业务对象进行任务调度也好，对业务对象通过远程公开使用也好，业务对象实现期间根本就不需要关心。当这些需求涌现的时候，可以在业务对象的基础上为它们提供相应的Wrapper或者Facade。让这些Wrapper或者Facade帮助当前业务对象独当一面，只有真正需要业务逻辑的时候，再将实质性的请求委派给业务对象处理。

我想，Spring Remoting带给我们的不只是一套现成的框架类以供我们使用。从一个侧面上来看，它向我们传达的是一种理念，以业务逻辑为中心进行开发的理念，坚持这种理念，整个系统的演化将变得不再举步维艰。

33.3.1 基于RMI的Remoting方案

RMI是Java SE平台标准的远程机制，它采用Java的序列化（Java Serialization）机制进行远程数据传输。Spring Remoting对现有的RMI使用进行了封装，在支持传统的RMI编程的基础上，还引入了RMI Invoker机制来进一步解除业务对象与RMI使用之间的耦合。

1. 通过RMI公开远程服务

按照RMI的要求，要想将某个服务通过RMI公开出去，它的接口定义必须符合某些要求。比如，接口定义需要继承java.rmi.Remote，接口中需要公开的操作对应的方法定义需要抛出Remote-Exception异常。所以，要想将我们的ITTMRateService以RMI公开出去，看来得对其做点儿手脚了，如下所示：

```
public interface ITTMRateService extends Remote{
  List<TTMRate> getTTMRatesByTradeDate(TradeDate tradeDate) throws RemoteException;
}
```

另外，业务接口中定义的方法参数和返回值都必须符合Java序列化的要求。因为我们的TradeDate使用了JodaTime的DateTimeFormatter，该类不符合要求，所以，我们暂且将其定义为transient，以避免序列化问题，如下所示：

```
private final transient DateTimeFormatter defaultDateTimeFormatter = DateTimeFormat. ➥
forPattern("yyyyMMdd");
```

在业务接口和实现类都符合RMI的要求之后，我们就可以通过Spring Remoting的org.springframework.remoting.rmi.RmiServiceExporter将其公开出去了，如代码清单33-1所示。

代码清单33-1 通过Spring的RMI支持公开远程服务配置代码示例

```
# server-config.xml
<bean id="ttmService" class="..MockTTMRateService">
</bean>

<bean id="ttmServiceWithRMIRemoting"
  class="org.springframework.remoting.rmi.RmiServiceExporter">
  <!-- with serviceName, we can access it with URL like [rmi://host:port/serviceName] -->
  <property name="serviceName" value="TtmService"/>
  <property name="serviceInterface" value="..ITTMRateService"/>
  <property name="service" ref="ttmService"/>
  <property name="registryPort" value="1099"/>
</bean>
```

现在，RMI客户端可以从rmi://yourhost:1099/TtmService对我们的ITTMRateService服务进行访问。

应该说，RMI需要相应的业务接口去实现Remote接口，以及抛出RemoteException等要求并不过分。毕竟，要在茫茫“类”海中寻找需要RMI服务的类终究需要某些标志吧？但是，将这些要求直接加诸到我们的业务接口定义以及实现类上，那就是我们的不对了！我们已经说了，业务对象只需要关心业务逻辑相关的关注点，对于使用它的方式，它应该一概不管，可现在我们显然是逆之而行了。实际上，我们完全可以把RMI需要业务接口戴上的那顶帽子转扣在别人头上，找个经纪人而已嘛，当真正需要的时候再来找我们的业务对象也不迟啊！

Spring Remoting引入了RMI Invoker的机制，来避免Remote接口以及RemoteException对业务接口和相应实现类的侵入，所以，如下这样保持我们最初的ITTMRateService定义不变：

```
public interface ITTMRateService{
  List<TTMRate> getTTMRatesByTradeDate(TradeDate tradeDate);
}
```

使用同样的RmiServiceExporter，同样的配置，我们依然可以将ITTMRateService以RMI的形式公开给相应客户端使用。只不过，RmiServiceExporter现在使用的是Spring提供的RMI Invoker机

制，而不是最初的RMI经典方式。

对于RMI Invoker的运作机制来说，总的原则就是把原来扣在ITTMRateService头上的RMI要求的那顶帽子，转扣在别人的头上。在Spring Remoting中，就是RemoteInvocationHandler，其定义如下：

```
public interface RmiInvocationHandler extends Remote {
   public String getTargetInterfaceName() throws RemoteException;

   public Object invoke(RemoteInvocation invocation) ➥
   throws RemoteException, NoSuchMethodException, IllegalAccessException, ➥
   InvocationTargetException;
}
```

可以看到，Remote接口和RemoteException现在全部扣在了RmiInvocationHandler的头上。当RmiServiceExporter检测到业务接口没有符合RMI要求的时候，它就会使用RMI Invoker机制，从invoke(RemoteInvocation)方法的参数中获取被调用的业务方法信息，然后通过反射调用目的业务对象的方法。这也就是业务接口得以从RMI的束缚中解脱的全部奥秘了。

因为RMI Invoker使用的是特定的传输对象（RemoteInvocation），所以对RMI Invoker公开的业务对象，只能通过Spring Remoting提供的RmiProxyFactoryBean进行访问。而对于按照RMI要求公开的服务接口，通过RmiServiceExporter公开之后，除了可以使用RmiProxyFactoryBean进行访问之外，普通的RMI客户端也同样可以访问。

2. 通过RMI访问远程服务

假设我们有依赖ITTMRateService的客户端对象定义如下：

```
public class TTMServiceClient {
private ITTMRateService ttmService;

   public void doSth(){
      List<TTMRate> ttmRates = ➥
         getTtmService().getTTMRatesByTradeDate(TradeDate.valueOf("20080302"));
      ...
   }
   // getter和setter方法定义……
}
```

如果像当初那样，TTMServiceClient与ITTMRateService实现类同处一地，我们可以直接将ITTMRateService实现类注入给它。但是，现在它们身居两地，我们就得另寻他法了。好在ttmService已经通过RMI以远程服务的形式公开了出来，我们可以使用org.springframework.remoting.rmi.RmiProxyFactoryBean来访问它。在RmiProxyFactoryBean创建了一个ITTMRateService的代理对象之后，我们就可以鱼目混珠，以ITTMRateService的代理对象代替实质的ITTMRateService实现类注入给TTMServiceClient使用，如下代码所示：

```
# client-config.xml
<bean id="ttmService" ➥
class="org.springframework.remoting.rmi.RmiProxyFactoryBean">
   <property name="serviceUrl" value="rmi://your host:1099/TtmService"/>
   <property name="serviceInterface" value="..ITTMRateService"/>
</bean>

<bean id="client" class="..TTMServiceClient">
   <property name="ttmService" ref="ttmService"/>
</bean>
```

RmiProxyFactoryBean使得TTMServiceClient对ITTMRateService的依赖变得透明。虽然现在

的ITTMRateService是以远程调用方式进行访问的，可是TTMServiceClient却是浑然不知，或许依然以为是在以本地调用的方式使用ITTMRateService呢！

33.3.2 基于 HTTP 的轻量级 Remoting 方案

Spring Remoting提供了三种基于HTTP的远程方案：

- Spring框架自行开发的Http Invoker方案；
- 基于Caucho的Hessian远程方案；
- 基于Caucho的Burlap远程方案。

在进一步了解每一种方案之前，我们先来从总体上分析一下，这些基于HTTP的轻量级远程方案都有哪些共通之处，或者说它们是怎么实现的。

1. 基于HTTP远程方案的实现原型分析

实际上要理解基于HTTP的远程方案是如何实现的并不困难，即使你之前没有开发过任何Web应用程序，一旦完成了Spring MVC之旅，我就可以十分肯定地告诉你，你实际上已经使用基于HTTP的远程方案很久了！难道你通过浏览器这种特定的客户端所请求的，不就是远程的服务器上的各种服务吗？

要将一个本地服务以HTTP的方式公开给远程客户端使用（当然，并不仅限于浏览器），最简单的办法就是为其提供一个Servlet实现，让该Servlet统一接收客户端调用请求，然后根据请求调用本地的服务对象，并最终将调用结果返回给相应的客户端。不过，要为每个服务对象都提供一个Servlet实现显然有些冗余了，而且，我们通常也无法在Servlet中直接获取到Spring的IoC容器的各种服务。鉴于Spring MVC已经有现成的DispatcherServlet可用的情况下，要公开相应的服务对象，实际上只需要提供相应的Controller即可。哦，确切地讲，是Handler。因为Spring Remoting为各种基于HTTP的远程方案所提供的Service Exporter实现，实际上都属于另一种类型的Handler，即HttpRequestHandler。该接口定义如下：

```
public interface HttpRequestHandler {
   void handleRequest(HttpServletRequest request, HttpServletResponse response) throws ➥
   ServletException, IOException;
}
```

现在，我们完全可以想象出，Spring Remoting到底是如何通过各种基于HTTP的轻量级远程方案将相应的服务对象公开出去的。正如代码清单33-2中的原型代码所展示的那样。

代码清单33-2 用于HTTP Remoting的HttpRequestHandler原型代码示例

```
public class PrototypeServiceExporter implements HttpRequestHandler {
   private IYourServiceType yourService;

   public void handleRequest(HttpServletRequest request, HttpServletResponse response) ➥
   throws ServletException, IOException {
      Object requestData = getMethodNameAndArgumentsOfServiceMethod(request);
      Object result = invokeServiceMethod(yourService,requestData);
      writeInvocationResultToClient(result,response);
   }
   // setter方法、getter方法和其他helper方法定义……
}
```

当把类似的PrototypeServiceExporter添加到DispatcherServlet的上下文之后，如下所示：


```
<bean name="/yourServiceExposedViaHttp" class="..PrototypeServiceExporter">
```

```
  <property name="yourService" ref="yourService"/>
</bean>
```

我们是不是就可以通过如下的URL进行访问了呢？

```
http://host:port/application/yourServiceExposedViaHttp
```

当然，现在我们通过该URL发送的请求内容，要变成所调用的远程服务的方法名以及相关的调用参数了。

通过HTTP公开远程服务实际上就是这么回事，写写相应的Handler而已。不过，使用Spring Remoting之后，这些Handler都不用我们写，因为Spring Remoting已经提供了现成的实现，如图33-6所示。

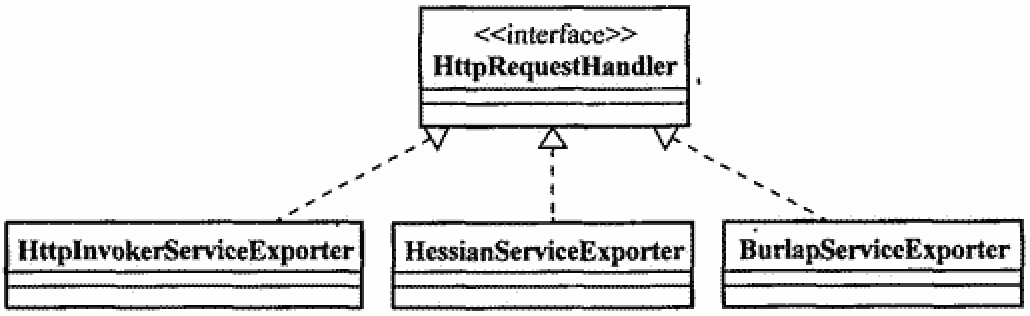

图33-6　基于HTTP的Service Exporter示意图

我们所要做的，就剩下在使用的时候简单配置一下它们了！

对于相应客户端如何访问各种基于HTTP的远程服务的处理方式，Spring Remoting采用的是ProxyFactoryBean的形式。这样做的好处我们已经了解了，相应的ProxyFactoryBean负责将远程服务接口的代理对象注入给客户端，这样，客户端调用方就可以透明的方式访问这些服务。在实现细节上，不用说也能猜得到。当调用方调用远程服务代理对象的相应方法的时候，ProxyFactoryBean挂钩的MethodInterceptor就可以对调用进行拦截，然后将拦截到的方法调用信息编组发送到服务端。至于将请求数据序列化并发送的功能实现，这就有点儿“八仙过海，各显神通”的味道了。虽然请求的数据内容基本相同，基本上都是要调用的方法名和相应方法参数，但因为各家所采用的数据序列化机制不同，在发送这些请求数据的时候，就得区别对待。对于HTTP Invoker来说，它采用的是标准的Java序列化机制，所以，我们可以通过数据流将请求数据发送给服务器端。而对于Hessian和Burlap来讲，因为数据的发送和编组已经有相应的类负责了，比如对应Hessian的HessianProxyFactory，Spring Remoting的Service Accessor实现只是借用一些这些现成的实现类来发送请求数据而已。

基本上，基于HTTP的远程方案只有这些内容。下面让我们从理论转入实践部分，看一下具体使用相应的HTTP远程方案是如何操作的吧！

2. Spring HTTP Invoker

Spring的HTTP Invoker采用标准的Java序列化机制。这与基于RMI的远程方式相同，如果你使用RMI过程中遭受防火墙之类因素的困扰，又不忍舍弃使用功能强大的标准Java数据序列化机制的好处，那么HTTP Invoker或许应该成为你考虑的对象。

要通过HTTP Invoker机制公开一个本地服务，我们使用org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter。但是既然它是一个HttpRequestHandler，要想让它开始工作，我们首先需要将它挂接到某个DispatcherServlet上。如果你有现成的Spring MVC应用所使用的DispatcherServlet，那么，使用现成的也可以。不过，重新设立一个DispatcherServlet专门负责HTTP形式的远程调用，或许能让整个应用看起来更“整齐”一些。所以，我们不妨在现有Web应用的配置文件web.xml中，新添加一个DispatcherServlet用于远程调用（见代码清单33-3）。

代码清单33-3　用于Remoting的DispatcherServlet配置示例

```
# web.xml
...
<servlet>
   <servlet-name>remoting</servlet-name>
   <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
   <load-on-startup>4</load-on-startup>
</servlet>

<servlet-mapping>
   <servlet-name>remoting</servlet-name>
   <url-pattern>/remoting/*</url-pattern>
</servlet-mapping>
...
```

有了它之后，我们就可以将ITTMRateService对应的HttpInvokerServiceExporter添加到该DispatcherServlet对应的ApplicationContext中，如下代码所示：

```
# remoting-servlet.xml
<bean name="/ttmServiceViaHttpInvoker"
class="org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter">
   <property name="serviceInterface" value="..ITTMRateService"/>
   <property name="service" ref="ttmRateService"/>
</bean>
```

假设我们已经将ITTMRateService的实现类添加到了Web应用的顶层ApplicationContext中。在这些全部完备之后，相应的远程调用客户端应该可以从http://host:port/application/remoting/ttmServiceViaHttpInvoker访问到我们通过HTTP Invoker公开出去的ITTMRateService服务。

说到调用客户端如何访问已经通过HTTP Invoker远程公开出来的ITTMRateService服务，那就更简单了。还记得我们的TTMServiceClient吗？它可是依赖于ITTMRateService接口。显然，我们只要为其注入使用HTTP Invoker进行远程访问的ITTMRateService类型代理对象就可以了，而HttpInvokerProxyFactoryBean就是做这个工作的。代码清单33-4给出的正是HttpInvokerProxyFactoryBean的使用配置示例。

代码清单33-4　HttpInvokerProxyFactoryBean的使用配置示例

```
# client.xml
<bean id="ttmServiceViaHttpInvoker" class="org.springframework.remoting.httpinvoker. ➥
HttpInvokerProxyFactoryBean">
   <property name="serviceUrl" value="http://your ➥
   host:8080/application/remoting/ttmServiceViaHttpInvoker"/>
   <property name="serviceInterface" value="..ITTMRateService"/>
</bean>

<bean id="client" class="..TTMServiceClient">
   <property name="ttmService" ref="ttmServiceViaHttpInvoker"/>
</bean>
```

显然，是使用RMI还是HTTP Invoker进行远程访问，对于我们的TTMServiceClient来说，完全没有太大的影响。最终决定使用什么机制来访问远程公开的某个服务，唯一要做的，也就是为其提供Spring Remoting已经实现的各种不同的ProxyFactoryBean型的Service Accessor而已。

3. Caucho Hession 和 Burlap

Hessian和Burlap分别采用二进制形式和XML文本形式进行数据传输过程中的序列化，而非采用标准的Java序列化机制。不过，这并不妨害二者的优秀表现。实际上，Caucho已经为我们提供了通过

Hessian和Burlap公开和访问相应远程服务的支持，但是，在我们使用Spring框架进行开发过程中，这些支持无法获取像依赖注入之类的服务，比如，Hessian和Burlap需要为每一需要公开的服务对象都提供对应的Servlet实现。显然，这些实现是无法获得依赖注入的好处的，所以，Spring Remoting通过DispatcherServlet统一了这片土地。在HessianServiceExporter和BurlapServiceExporter以HttpRequetHandler的身份出现的时候，它们自然获得了Spring框架的各种服务。

使用Hessian和Burlap来公开我们的ITTMRateService给相应远程客户端使用，分别使用Hessian-ServiceExporter和BurlapServiceExporter这两个Service Exporter实现。与基于HTTP Invoker的远程服务公开类似，如果最初没有DispatcherServlet为我们的HessianServiceExporter和BurlapSer-viceExporter提供服务支持的话，请先行准备之。在相应的DispatcherServlet就绪之后（我们不妨沿用HTTP Invoker使用的DispatcherServlet），我们只要将HessianServiceExporter和BurlapSer-viceExporter添加到其ApplicationContext中，即可激活ITTMRateService基于Hessian和Burlap的对外远程服务，如代码清单33-5所示。

代码清单33-5　通过HessianServiceExporter和BurlapServiceExporter远程公开服务配置示例

```
# remoting-servlet.xml
<bean name="/ttmServiceViaHessian" ➥
class="org.springframework.remoting.caucho.HessianServiceExporter">
   <property name="serviceInterface" value="..ITTMRateService"/>
   <property name="service" ref="ttmRateService"/>
</bean>

<bean name="/ttmServiceViaBurlap" ➥
class="org.springframework.remoting.caucho.BurlapServiceExporter">
   <property name="serviceInterface" value="..ITTMRateService"/>
   <property name="service" ref="ttmRateService"/>
</bean>
```

只要业务对象设计良好，本地调用自然没的说，就是要去远程公开它，现在也只是添加相应的“外围设备”而已。

至于对HessianServiceExporter和BurlapServiceExporter公开出来的ITTMRateService服务，我们使用对应的HessianProxyFactoryBean和BurlapProxyFactoryBean对其进行访问。既然TTMServiceClient依赖于远程接口，我们所要做的也只是将HessianProxyFactoryBean和Burlap-ProxyFactoryBean注入给它了，如代码清单33-6所示。

代码清单33-6　HessianProxyFactoryBean和BurlapProxyFactoryBean使用配置示例

```
# client.xml
<bean id="ttmServiceViaBurlap" ➥
class="org.springframework.remoting.caucho.BurlapProxyFactoryBean">
   <property name="serviceUrl" value="http://yourhost:8080/application/remoting/ ➥
   ttmServiceViaBurlap"/>
   <property name="serviceInterface" value="..ITTMRateService"/>
</bean>

<bean id="ttmServiceViaHessian" ➥
class="org.springframework.remoting.caucho.HessianProxyFactoryBean">
   <property name="serviceUrl" ➥
   value="http://yourhost:8080/application/remoting/ttmServiceViaHessian"/>
   <property name="serviceInterface" value="..ITTMRateService"/>
```

```
</bean>

<bean id="client" class="..TTMServiceClient">
  <property name="ttmService" ref="ttmServiceViaHessian 或者 ttmServiceViaBurlap"/>
</bean>
```

客户端到底要通过什么样的方式来访问远程服务，看来只是简单的配置更改了。

> **注意** 有关Hessian、Burlap和Http Invoker的更多特性，比如安全验证之类，也可以通过相应Service Exporter/Accessor的配置项进行定义。实际需要的时候，不妨直接查阅对应的Javadoc文档，我们就不对这部分内容做过多阐述了。

33.3.3 基于Web服务的远程方案

Spring Remoting从最初就对JAX-RPC标准的Web服务提供支持。顺应JAX-RPC的后继者JAX-WS后浪推前浪之势，Spring在2.5发布之后，又添加了对基于JAX-WS标准的Web服务的相应支持。但不管采用什么样的Web服务标准，在Spring中的访问风格基本都是一致的，依然是抽象概念上的Service Exporter和Service Accessor，只不过具体的实现细节不同而已。

1. 通过Web服务公开远程服务

如果说Spring为其他远程方案提供的Service Exporter和Service Accessor实现都是均衡发展的话，那么，对Web服务的支持就有些"偏袒"的意味了。在使用Web服务以远程方式公开本地服务方面，Spring并没有像其他远程方案那样，为其提供完备的Service Exporter实现，除了一个简单的`ServletEndpointSupport`可供我们驱使之外，标准的Spring框架内再没有相关的更多支持。出现这种状况的原因如下所述。

- 要实现一套完备的Web服务开发框架需要付出很多的努力。在Spring Remoting框架内实现一套完备的Web服务开发框架，显然已经超出了Spring Remoting只是希望提供轻量级远程方案的本意。不过，虽然不能将一套完备的Web服务开发框架直接放到Spring框架内发布（可能带来开发、管理和维护上的问题），但是，却完全可以使用Spring框架的理念单独在次一级的项目中开发一套Web服务框架。属于Spring框架子项目的Spring Web Services（http://static.springframework.org/spring-ws/site/）恰好就提供了这么一套Web服务开发框架。
- 现在已经存在许多成熟的Web服务开发框架，除了Spring Web Services项目，还有Apache的Axis/Axis2、Codehaus的xFire（http://xfire.codehaus.org/），以及现在合并了xFire和Celtix孵化完成的Apache CXF（http://cxf.apache.org/）项目。但各种Web服务开发框架的实现，在公开具体服务的时候都有特定于自己的方式，所以，看来Spring框架在统一这一问题的处理方面确实也有些无能力为了。

虽然Spring没有为以Web服务形式远程公开相应服务提供太多支持，但我们通常可以使用相应Web服务开发框架提供的各种手段，尤其像xFire和CXF，它们与Spring框架可以很好地集成。我们不妨以CXF为例，看一下它是如何将本地业务对象以Web服务的形式公开出去的。

> **注意** `org.springframework.remoting.jaxrpc.ServletEndpointSupport`为我们实现JAX-RPC中Servlet类型的Endpoint提供了便利支持，可以让我们在实现类中获取Spring的`ApplicationContext`容器的支持。如果需要的话，可以扩展该类实现自己的Servlet类型EndPoint。Spring参考文档中就`ServletEndpointSupport`的使用提供了简短的介绍。

要借助于CXF将ITTMRateService以Web服务的形式公开为远程服务，我们可以使用CXF提供的org.apache.cxf.jaxws.JaxWsServerFactoryBean实现类。在将我们的ITTMRateService以及相关类如下所示稍作手脚之后：[①]

```
@WebService
public interface ITTMRateService {
   List<TTMRate> getTTMRatesByTradeDate(TradeDate tradeDate);
}
```

我们就可以通过向服务端的ApplicationContext中添加代码清单33-7所示的配置内容，将ITTMRateService以Web服务的形式公开出去。

代码清单33-7　通过CXF公开Web服务的配置代码示例

```
<beans xmlns="http://www.springframework.org/schema/beans" ➥
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ➥
xmlns:jaxws="http://cxf.apache.org/jaxws" ➥
xsi:schemaLocation="http://www.springframework.org/schema/beans ➥
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd ➥
http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd">

<import resource="classpath:META-INF/cxf/cxf.xml" />
<import resource="classpath:META-INF/cxf/cxf-extension-soap.xml" />
<import resource="classpath:META-INF/cxf/cxf-servlet.xml" />

<jaxws:endpoint id="ttmServiceViaCXF" address="/ttmServiceViaCXF" ➥
implementor="#ttmRateService">
   </jaxws:endpoint>

   <bean id="ttmRateService" class="..service.MockTTMRateService">
   </bean>
   ...
</beans>
```

<jaxws:endpoint>的背后是JaxWsServerFactoryBean在支撑，而JaxWsServerFactoryBean在这里实际上就是我们的Service Exporter，所以，CXF从总体理念上都是与Spring的Remoting理念保持看齐的。[②]

注意　实际上，要让我们的ITTMRateService真正以Web服务公开出去，我们还有部分细节需要处理。按照CXF的配置要求，如果要通过HTTP形式将当前服务公开出去的话，我们需要向web.xml添加CXFServlet相关定义，并让Spring的ContextLoaderListener加载我们上面的ApplicationContext配置文件，具体细节请参考CXF配置文档。

2. 通过Web服务访问远程服务

要访问刚才通过CXF公开出来的Web服务，最直接的方式就是使用CXF同时提供的Service Accessor实现，即org.apache.cxf.jaxws.JaxWsProxyFactoryBean，而且，在CXF提供了自定义命名空间的情况下，我们可以很容易地在客户端的ApplicationContext配置文件中添加该Service

① 我们的TradeDate和TTMRate都需要做相应改动，以满足相应的要求，该做哪些手脚，请参考JAX-WS规范文档，这跟Spring无关。

② <jaxws:endpoint>的implementor属性通常指定实现类，如果要引用现有的bean定义的话，前缀#，我们恰好是这么做的。

Accessor的定义并使用，如代码清单33-8所示。

代码清单33-8 通过CXF访问Web服务配置代码示例

```
<beans xmlns="http://www.springframework.org/schema/beans" ➥
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ➥
xmlns:jaxws="http://cxf.apache.org/jaxws" ➥
xsi:schemaLocation="http://www.springframework.org/schema/beans ➥
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd ➥
    http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd">

  <import resource="classpath:META-INF/cxf/cxf.xml" />
  <import resource="classpath:META-INF/cxf/cxf-extension-soap.xml" />
  <import resource="classpath:META-INF/cxf/cxf-servlet.xml" />

  <bean id="client" class="..TTMServiceClient">
    <property name="ttmService" ref="ttmServiceViaCXF"/>
  </bean>

  <jaxws:client id="ttmServiceViaCXF" address="http://your ➥
  host:8080/application/cxf/ttmServiceViaCXF" serviceClass="..ITTMRateService">
  </jaxws:client>

</beans>
```

我想，`<jaxws:client>`背后的奥秘现在看来应该不难理解吧？

除了使用像CXF之类Web服务框架提供的特定的Service Accessor实现来访问相应远程服务之外，我们还可以使用Spring Remoting专门为JAX-RPC和JAX-WS类型Web服务访问提供的“制式装备”，详情如表33-1所列的。

表33-1 Spring为Web服务提供的Service Accessor实现

WS类型	Service Accessor
JAX-RPC	org.springframework.remoting.jaxrpc.LocalJaxRpcServiceFactoryBean org.springframework.remoting.jaxrpc.JaxRpcPortProxyFactoryBean
JAX-WS	org.springframework.remoting.jaxws.LocalJaxWsServiceFactoryBean org.springframework.remoting.jaxws.JaxWsPortProxyFactoryBean

对于同样的ITTMRateService远程服务，我们可以选择使用JaxWsPortProxyFactoryBean完成同样的访问。不过这次看起来好像没有CXF“量身定做”的Service Accessor那么合身罢了，如代码清单33-9所示。

代码清单33-9 通过JaxWsPortProxyFactoryBean访问Web服务配置代码示例

```
<bean id="client" class="..TTMServiceClient">
  <property name="ttmService" ref="ttmServiceViaCXF"/>
</bean>

<bean id="ttmServiceViaCXF" class="org.springframework.remoting.jaxws.JaxWsPortProxyFactoryBean">
  <property name="serviceInterface" value="..ITTMRateService"/>
  <property name="wsdlDocumentUrl" value="http://localhost:8080/simplefx/cxf/ttmService ➥
  ViaCXF?WSDL"/>
  <property name="namespaceUri" value="http://service.simplefx.spring21.cn/"/>
  <property name="serviceName" value="MockTTMRateServiceService"/>
  <property name="portName" value="MockTTMRateServicePort"/>
</bean>
```

33

JaxWsPortProxyFactoryBean定义的各种属性，比如namespaceUrl、serviceName和portName分别对应.wsdl文件中的targetNamespace、服务名称以及端口名称。我想，通过wsdlDocumentUrl访问到.wsdl文件之后，这些信息不难找到（代码清单33-9中是我的本地应用对应的各种属性值）。

> **注意**　俗话说得好，“功夫在诗外”，要发挥Spring Remoting提供的这些Web服务相关的Service Accessor实现类的功效，除了参考相应的Javadoc之外，我想，我们不得不首先去理解JAX-RPC和JAX-WS以及其他Web服务规范吧？

目前为止，我们说的其实都是一个问题，那就是，不管我们使用的Service Accessor实现类怎么变换，客户端调用代码却可以“我自岿然不动”，唯一变动的只是简单的配置变换而已。我想Spring Remoting的Service Accessor的魅力就在这里。所以，如果某些情况下，Spring Remoting没有为我们提供相应的Service Accessor实现的话，就算我们“依葫芦画瓢”开发一个也不是什么难事吧？当然，如果大多数远程方案都能够像CXF那样，事先提供现成的Service Accessor实现类，那就实在太好了。

33.3.4　基于 JMS 的远程方案

要说Spring中基于JMS的远程方案，那一定要提到Codehaus的Lingo（http://lingo.codehaus.org/Home）。早在Spring 2.5发布了对基于JMS的远程方案的官方支持之前，Lingo就已经活跃在战斗第一线了。Lingo的远程消息架构完全遵循Spring Remoting的Service Exporter和Service Accessor统一实现风格，而且，提供了对各种远程消息模式的支持。即使是在Spring 2.5引入了官方标准JMS远程方案之后，Lingo依然是值得我们关注的。当然，既然我们主要关注的是Spring Remoting标准方案，下面我们还是要围绕着Spring Remoting提供的基于JMS远程方案进行阐述。现在让我们开始吧！

> **注意**　Lingo拥有许多Spring Remoting标准JMS远程方案所没有的特性，你可以通过http://lingo.codehaus.org/Home以获取Lingo的更多信息，“工欲善其事，必先利其器”，如果Spring Remoting提供的标准JMS远程方案无法满足我们的当前场景需求，不妨转而拜访Lingo试试。

1. 基于JMS的远程方案实现分析

在我们已经了解了Spring对JMS的集成相关内容之后，要理解Spring Remoting提供的基于JMS的远程方案也就要简单多了。我们开门见山吧！

在JMS场景中，两个不同的对象或者组件需要相互通信，基本上都是通过指定的Destination作为“桥梁”。在同步场景中，客户端对象发送消息到指定的Destination，服务端对象即可从同样的Destination中接收客户端发送的消息进行处理。如果处理需要返回结果，那么服务端对象通常将结果以JMS消息的形式发送到请求消息的JMSReplyTo指定的Destination中。这样，客户端即可从其当初为请求消息设置的JMSReplyTo对应的Destination中获取处理结果消息。实际上，Spring Remoting提供的基于JMS的远程方案运作方式与此基本相同，只不过，发送的消息形式是固定的。在通常的同步场景中，我们或许可以发送任何希望的消息类型，而在Spring Remoting的基于JMS的远程方案中，我们发送的消息类型为对应RemoteInvocation类型的ObjectMessage。

在客户端通过指定的服务接口调用相应的业务方法之后，对应的Service Accessor将通过MethodInterceptor对服务接口上的业务方法调用进行拦截，然后将调用信息（通常是方法名和相应的方法参数）以RemoteInvocation进行封装，并创建为ObjectMessage以发送。但是，发送请求消息之前，Service Accessor有一件事情要做，那就是创建一个Temporary Queue以作为调用结果返回的Destination。

在Temporary Queue创建之后，把它设置为要发送的ObjectMessage的JMSReplyTo就可以开始发送了。

公开远程服务的Service Exporter将从与Service Accessor发送消息使用的同一个Destination中接收请求消息，然后对实际消息类型为RemoteInvocation的ObjectMessage进行解码，提取RemoteInvocation中的调用信息，并通过反射调用本地业务对象对应的业务方法。在调用完成之后，ServiceExporter将同样对调用结果进行封装（封装类为RemoteInvocationResult），最后，Service Exporter会将封装后的调用结果消息发送到请求消息的JMSReplyTo指定的Destination，我们在发生消息的时候已经创建它了。

在Service Accessor以同步的形式从其最初创建的Temporary Queue中接收到调用结果消息之后，即可把消息解码然后返回给客户端调用对象，至此，整个过程即告完成。

我们可以使用图33-7来进一步描述整个远程服务的公开和访问过程。

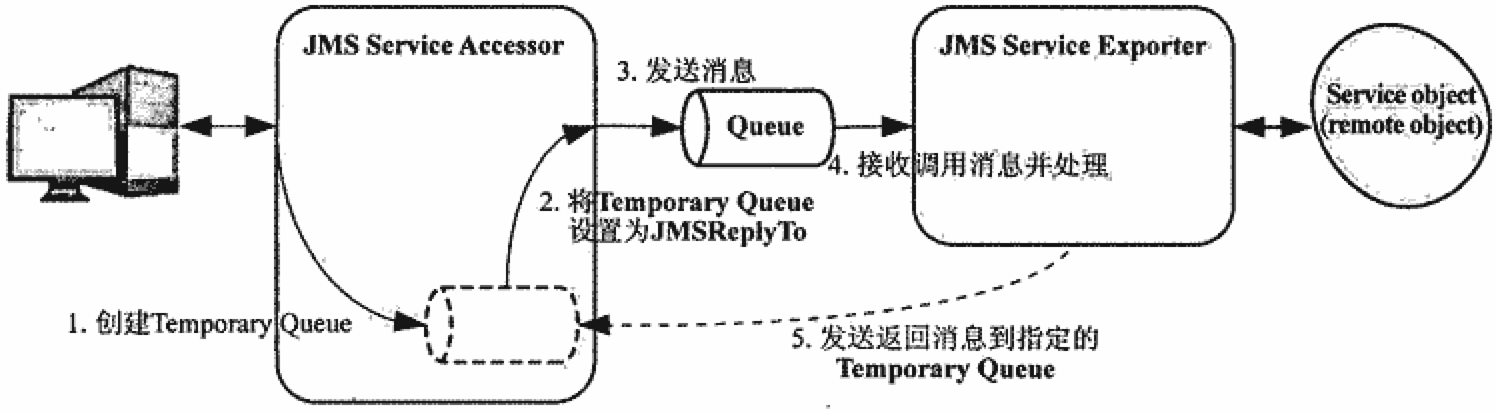

图33-7 JMS Remoting方案实现原理示意图

实际上，图33-7所展示的情景跟我们使用Temporary Queue完成JMS同步消息发送和接收的结构图没有什么差别，不是吗？

2. 通过JMS公开和访问远程服务

在了解了Spring Remoting中基于JMS的远程方案实现原理之后，要运用就更加轻松了。

在开始正式工作之前，我们先设置一些客户端和服务端都要使用的共同设施，比如ConnectionFactory和二者共同使用的Destination。我们将这些公用设施放到一个单独的Spring IoC容器配置文件中，如下所示：

```
# jms-remoting-commons.xml
<util:properties id="jndiEnv" location="classpath:amq_jndi.properties"/>
<jee:jndi-lookup id="connectionFactory" jndi-name="ConnectionFactory" environment-ref="jndiEnv"/>
<jee:jndi-lookup id="remotingQueue" jndi-name="dynamicQueues/remoting" environment-ref="jndiEnv"/>
...
```

我们这里依然使用ActiveMQ，但并不是必要的，你也可以选择其他的JMS Provider实现，这完全要根据实际情况来决定。

在有了基础设施之后，我们开始通过JMS公开我们的ITTMRateService。在基于JMS的远程方案中，对应的Service Exporter实现是org.springframework.jms.remoting.JmsInvokerServiceExporter。它实际上是一个消息驱动POJO，因为它实现了SessionAwareMessageListener接口（希望你还记得它与MessageListener的区别）。所以，要使用JmsInvokerServiceExporter公开ITTMRateService服务，看来只需要将它挂接到相应的MessageListenerContainer就好，如代码清单33-10所示。

33

代码清单33-10　通过JmsInvokerServiceExporter公开ITTMRateService配置代码示例

```
# server.xml
<bean id="ttmRateService" class="..MockTTMRateService">
</bean>

<bean id="ttmServiceExporter" ➥
class="org.springframework.jms.remoting.JmsInvokerServiceExporter">
   <property name="serviceInterface" value="..ITTMRateService"/>
   <property name="service" ref="ttmRateService"/>
</bean>

<bean ➥
class="org.springframework.jms.listener.DefaultMessageListenerContainer">
   <property name="connectionFactory" ref="connectionFactory"/>
   <property name="destination" ref="remotingQueue"/>
   <property name="concurrentConsumers" value="3"/>
   <property name="messageListener" ref="ttmServiceExporter"/>
</bean>
```

现在，JmsInvokerServiceExporter可以迎接任何对ttmRateService的调用请求了。

小心　我们使用通常的bean定义形式来使用DefaultMessageListnerContainer或者Simple-MessageListenerContainer，而没有使用新添加的jms命名空间下的<jms:listenercontainer>和<jms:listener>。因为这会导致服务端接收不到调用消息，具体原因我也不清楚，猜测可能是<jms:listener-container>和<jms:listener>的handler有部分细节没有处理造成的。

若要访问基于JMS的远程服务，我们使用的Service Accessor实现类为org.springframework.jms.remoting.JmsInvokerProxyFactoryBean，它既然要发送消息到指定的Destination，自然需要一些必要的装备设置，所以，通常提供如代码清单33-11所示的配置之后，JmsInvokerProxy-FactoryBean就可以将合适的ITTMRateService代理对象注入给客户端对象了。

代码清单33-11　通过JmsInvokerProxyFactoryBean访问远程服务配置代码示例

```
# client.xml
<bean id="client" class="..TTMServiceClient">
   <property name="ttmService" ref="ttmServiceViaJms"/>
</bean>

<bean id="ttmServiceViaJms" class="org.springframework.jms.remoting.JmsInvokerProxyFactoryBean">
   <property name="serviceInterface" value="..ITTMRateService"/>
   <property name="connectionFactory" ref="connectionFactory"/>
   <property name="queue" ref="remotingQueue"/>
</bean>
```

虽然我们数次替换掉了TTMServiceClient所使用的ITTMRateService，但“可怜却又如此舒服”的TTMServiceClient依然对于自己使用的ITTMRateService实现类在哪里是浑然未知，幸哉！

33.4　扩展 Spring Remoting

通常情况下，Spring Remoting提供的各种Service Accessor以及Service Exporter实现类已经可以满

足我们大部分的应用场景需要。即使默认设置不能满足需要，通过各Service Accessor和Service Exporter的相应属性设置也可以进一步定制它们的行为，但不可否认的是，总会有特殊的需求场景出现。这时，我们可以考虑引入自定义的RemoteInvocationFactory和RemoteInvocationExecutor来扩展Spring Remoting的相应远程方案。

Spring Remoting提供的基于RMI、HTTP Invoker和JMS的远程方案建立在几乎相同的实现结构上。用一句话来总结的话，整个实现结构就是“一个中心，两个基本点”，即以RemoteInvocation为中心，以RemoteInvocationFactory和RemoteInvocationExecutor为两个基本点。它们之间的关系如图33-8所示。

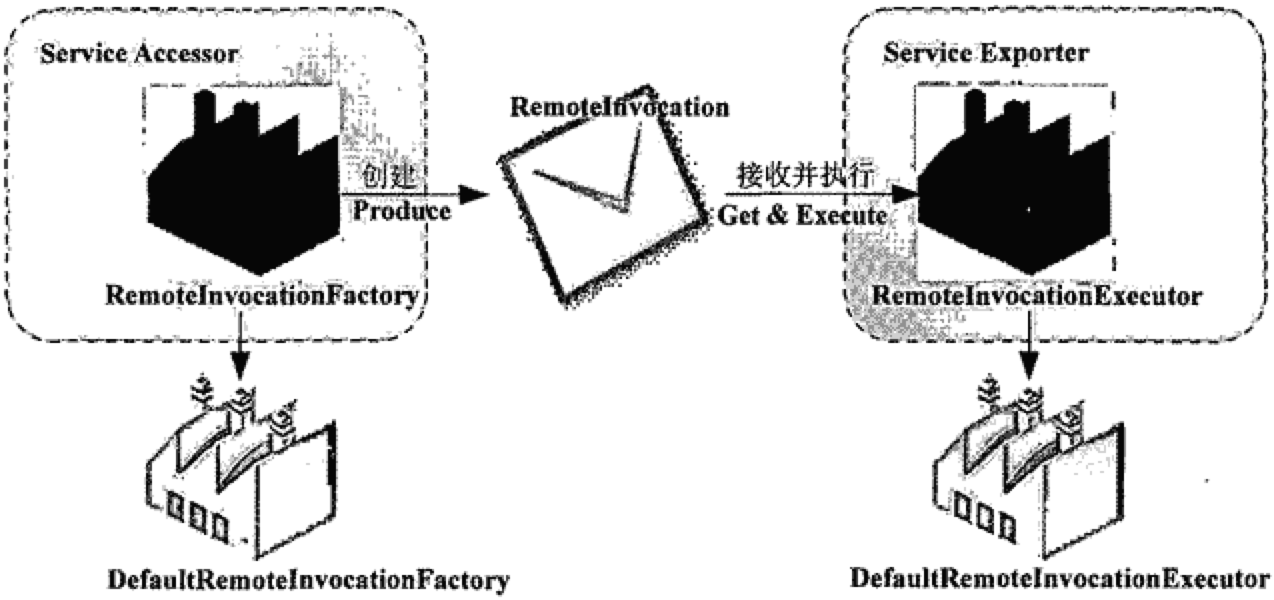

图33-8 Spring Remoting远程调用实现结构示意图

RemoteInvocation是整个实现结构的核心，它实际上就是一个数据对象，封装了远程调用上下文信息。从该类的定义我们可以更加清楚地了解它的“底细”，代码清单33-12为该类实现的部分代码摘录。

代码清单33-12 RemoteInvocation实现代码摘录

```
public class RemoteInvocation implements Serializable {
   ...
   private String methodName;
   private Class[] parameterTypes;
   private Object[] arguments;
   private Map attributes; // ***这里是我们的扩展点***

   // 构造方法定义……
   ...

   public void addAttribute(String key, Serializable value) throws IllegalStateException {
      if (this.attributes == null) {
         this.attributes = new HashMap();
      }
      if (this.attributes.containsKey(key)) {
         throw new IllegalStateException("There is already an attribute with key '" + key + "' bound");
      }
      this.attributes.put(key, value);
   }
```

33

```
    ...

    public Object invoke(Object targetObject) throws NoSuchMethodException, ➥
    IllegalAccessException, InvocationTargetException {
      Method method = targetObject.getClass().getMethod(this.methodName, this.parameterTypes);
      return method.invoke(targetObject, this.arguments);
    }
    ...
}
```

大多数情况下，RemoteInvocation传输的远程调用上下文只包括前面三项信息，即调用的方法名称（methodName）、调用的方法的参数类型（parameterType）以及调用的方法的参数值（arguments）。但是，我们可以通过RemoteInvocation的attributes属性添加更多自定义的调用信息，而这也正是我们可以使用的扩展点。

既然已经知道了RemoteInvocation主要是一个数据对象，我们下一步要寻找的目标当然就是RemoteInvocation中的数据是从何而来的。这也就引出了“两个基本点”之一的RemoteInvocationFactory。在基于RMI、Http Invoker和JMS的远程方案对应的Service Accessor，通过相应的MethodInterceptor拦截到对远程对象的方法调用之后，这些Service Accessor会把“将调用信息封装为RemoteInvocation”这样的工作交给相应的RemoteInvocationFactory去做，因为这是RemoteInvocationFactory的主要职责。其定义如下所示：

```
public interface RemoteInvocationFactory {
  RemoteInvocation createRemoteInvocation(MethodInvocation methodInvocation);
}
```

Spring Remoting为RemoteInvocationFactory只提供了一个默认实现类DefaultRemoteInvocationFactory。该类的行为只从createRemoteInvocation(MethodInvocation)方法的MethodInvocation参数中抽取到必要的方法名、方法参数类型和方法参数值信息，然后把这些信息添加到RemoteInvocation并返回。显然，如果我们想要RemoteInvocation携带更多信息，那么DefaultRemoteInvocationFactory看来是帮不上什么忙了，只能是自定义一个RemoteInvocationFactory实现。

RemoteInvocationFactory将远程调用信息以RemoteInvocation的形式返回给相应的Service Accessor之后，Service Accessor就可以将RemoteInvocation对象发送给相应的Service Exporter进行处理了。对应RMI、HTTP Invoker和JMS的Service Exporter实现接收到RemoteInvocation形式的远程调用信息之后，会将“根据RemoteInvocation封装的调用信息调用相应业务方法”的工作转交给一个RemoteInvocationExecutor，现在，“两个基本点”全部到齐了！

RemoteInvocationExecutor将根据RemoteInvocation的调用信息，调用指定的业务对象上对应的业务方法。最直接的实现当然是通过反射进行调用，而这也正是RemoteInvocationExecutor的默认实现类DefaultRemoteInvocationExecutor所做的。只不过，DefaultRemoteInvocationExecutor更懒，它直接把调用委派给了RemoteInvocation本身，RemoteInvocation定义的唯一行为invoke(Object)，恰好就是为RemoteInvocationExecutor准备的。在自定义的RemoteInvocationFactory向RemoteInvocation中添加了方法名、方法参数类型和方法参数值之外的调用信息之后，DefaultRemoteInvocationExecutor的默认行为看来是将无视这些信息了。所以相对的，如果我们提供了自定义的RemoteInvocationFactory，为了它提供的“多余”信息能够有人关注，那么也同时提供一个自定义的RemoteInvocationExecutor实现吧！

在自定义的RemoteInvocationFactory准备完毕之后，就可以通过相应Service Accessor的

setRemoteInvocationFactory(RemoteInvocationFactory)方法替换掉默认的RemoteInvocationFactory实现。相应地，对于自定义的RemoteInvocationExecutor来说，也同时可以通过相应Service Exporter实现类的setRemoteInvocationExecutor(RemoteInvocationExecutor)完成类似的替换过程。通过自定义RemoteInvocationFactory/RemoteInvocationExecutor来扩展Spring Remoting的整个过程就是这样。至于你要如何发挥它的作用，完全由你决定。最可能想到的可能就是将客户端的某些验证信息添加到RemoteInvocation，然后在远程服务端由相应的RemoteInvocationExecutor对这些信息进行处理，以决定相应服务的具体调用策略。至于更多的扩展场景，不妨放开你的思绪，让它尽情的飞翔吧！

小心　基于自定义RemoteInvocationFactory/RemoteInvocationExecutor的扩展方案，只适合于Spring Remoting提供的基于RMI、HTTP Invoker和JMS的远程方案。至于像基于Hessian/Burlap和Web服务的远程方案，因为它们有自己的传输机制和处理方式。要扩展它们，更多可能需要通过继承并覆写相应处理方法的方式进行。这当然要具体情况具体分析，如果能够通过外围设施，比如AOP，就能够解决相应问题的话，最好的扩展策略当然就是不扩展。

33.5 Spring Remoting之扩展篇

33.5.1 拉开JMX演出的序幕

在使用Spring进行应用开发的过程中，尤其是Standalone形式的应用程序，随着系统中类定义的增长，IoC容器启动速度将变得越来越慢。虽然Spring 2.5发布之后，对容器的启动性能进行了优化，使得这一时间消耗进一步减少，但某些场景下，这一问题仍然值得关注并加以解决。

为了加快IoC容器的启动时间，我们可以像下面这样做。

- 精简每次启动所需要加载的bean定义。比如在测试场景中，如果只用到一两个相关bean定义，那么只加载那些必须的bean定义即可。当然，这需要事先对bean定义的管理进行细分，以良好的组织结构对系统中所有的bean定义进行管理。
- 延迟加载相关的bean定义。默认情况下，ApplicationContext会在容器启动的时候将所有的bean都实例化，这是造成容器启动慢的一个原因。那么我们就可以尝试改变ApplicationContext的默认行为，让某些bean定义的实例化过程延后，甚至，让所有的bean定义都延迟加载。通过<bean>的lazy-load或者<beans>的default-lazy-load属性，我们可以达到这种目的。

除了像这些方法那样，从容器本身的特性进行考虑以改进启动速度，我们还可以从其他方面进行考虑，下面就是我曾经考虑的一种方式。

FX系统中的Rollover需要在特定的时间点关闭市场终止交易，对系统状态进行调整。Rollover完成之后，再开启市场，让用户可以继续进行外汇交易。如果规定的关闭市场时间是6:10，那么6:11分才关闭市场应该是不可容忍的。在这一前提下，我希望对Rollover程序做出改进，避免Spring的IoC容器启动时间过长进而导致市场关闭时间的延迟。我们的设想是这样的，既然每次启动应用程序都要有容器启动上的损害，我们就避免频繁启动，让整个Rollover应用程序以某种服务的形式存在。当需要关闭市场或者开启市场的时候，只需要运营人员通过某种远程机制将关闭和开启市场的请求发送给Rollover处理。大略上来说，如果远程调用上的时间损害能够小于容器启动上的时间损害，那么我们显然就赚到了。

我们已经了解了Spring Remoting提供的各种远程方案，但这次不打算使用它们（虽然它们也可以实现这一需求），因为我们认为这更贴近于对系统进行管理的场景，所以，JMX（Java Management Extension）将成为我们最终的选择。

JMX为我们对系统中的各种资源进行管理提供了统一的框架支持。随着各个产品对JMX的广泛应用，JMX正越来越受人瞩目，如果你还在以被动的形式去管理应用程序以及相应资源，那么现在该是借助于JMX转向主动防御的时候了。在你初步或者熟知JMX之后，希望你一鼓作气，接着踏上Spring中JMX相关旅程，你会发现，Spring为JMX开发带来的各种便利，同时也会发现许多是曾相识的身影，尤其是当JSR 160（JMX Remote API）相关的内容浮出水面的时候，Spring Remoting中的相关理念更应该跃入你的脑海。

遗憾的是，我不能继续陪各位踏上后继的旅程。“革命尚未成功，诸位仍需努力”，尝试着自己去挖掘Spring中的各种“宝藏”，或许你能够发现更多。希望你在完成Spring的JMX之旅之后，能够为我们的“容器启动慢”这一问题给出一个完美的解决答案。

33.5.2　Spring 3.0 展望

现有的Spring Remoting框架将随着技术和理念的发展而发展，在这一前提下，我们可以设想：

- 既然Spring 3.0将为Spring MVC添加REST支持，那么基于REST形式的远程调用是否也应该被给予足够重视呢？
- Web服务（Web Service）的开发将越来越向着更轻量级的方向发展，如果现有的基础装备不够完善，那我们将继续完善这些现有的基础装备；如果有新的理念出现，我们将奋起直追，添加相应的支持。

实际上，Spring 3.0也是要这么做的。

Spring 3.0将在推出针对SpringMVC的REST支持的同时，也将隆重推出REST服务访问的客户端支持，也就是`RestTemplate`。我可以把`RestTemplate`放到数据访问部分，因为它确实也可以属于那一类；我同样也可以把它放在Spring MVC部分进行说明，因为它跟REST息息相关；但合计来合计去，还是放在Remoting部分提到它合适一些，虽然它本身孤零零的，并不太符合Remoting框架的基本架构。不过，把`RestTemplate`和Spring Remoting中对Web服务的支持进行对比，多少还是有相似性的（它们都侧重于客户端访问支持）。从`RestTemplate`的名字也能猜到，它又是一个“模板方法模式”在Spring框架中的经典范例，至于实现细节，我想就不用赘述了吧？如果你对如何使用它很感兴趣，那倒是可以参考一下Spring团队的博客，http://blog.springsource.com/ 2009/03/27/rest-in-spring-3-resttemplate/，其中对`RestTemplate`的使用进行了详细的介绍。

在Web服务开发过程中，经常需要进行数据的编组和解组。要实现这一目的，我们有多种选择，比如JAXB、Castor、XStream等。Spring 3.0中将提供一个抽象层OXM（Object/XML Mapping）用来屏蔽这些不同解决方案之间的差异性，这将是一套全新的API，大家可以拭目以待哦！

这个世界上好像没有什么是不变的，唯一不变的是时间，但我们不能去跟时间抗衡。既然变化是不可避免的，那就让我们拥抱变化，拥抱新的Spring吧！